This Page Is Inserted by IFW Operations
and is not a part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problem Mailbox.

IMAGE EVALUATION TEST TARGET (MT-3)

150mm

6"

PHOTOGRAPHIC SCIENCES CORPORATION
770 BASKET ROAD
P.O. BOX 338
WEBSTER, NEW YORK 14580
(716) 265-1600

microunity

Zeus System Architecture

COPYRIGHT 1998 MICROUNITY SYSTEMS ENGINEERING, INC. ALL RIGHTS RESERVED

MicroUnity

Craig Hansen
Chief Architect

MicroUnity Systems Engineering, Inc.
475 Potrero Avenue
Sunnyvale, CA 94086.4118
Phone: 408.734.8100
Fax: 408.734.8136
email: craig@microunity.com
http://www.microunity.com

Store

These operations add the contents of two registers to produce a virtual address, and store the contents of a register into memory.

Operation codes

S.8[22]	Store byte
S.16.B	Store double big-endian
S.16AB	Store double aligned big-endian
S.16.L	Store double little-endian
S.16AL	Store double aligned little-endian
S.32.B	Store quadlet big-endian
S.32AB	Store quadlet aligned big-endian
S.32.L	Store quadlet little-endian
S.32AL	Store quadlet aligned little-endian
S.64.B	Store octlet big-endian
S.64AB	Store octlet aligned big-endian
S.64.L	Store octlet little-endian
S.64AL	Store octlet aligned little-endian
S.128.B	Store hexlet big-endian
S.128AB	Store hexlet aligned big-endian
S.128.L	Store hexlet little-endian
S.128AL	Store hexlet aligned little-endian
S.MUX.64AB	Store multiplex octlet aligned big-endian
S.MUX.64AL	Store multiplex octlet aligned little-endian

Selection

number format	op	size	alignment	ordering
byte		8		
integer		16 32 64 128		L B
integer aligned		16 32 64 128	A	L B
multiplex	MUX	64	A	L B

Format

op rd,rc,rb

op(rd,rc,rb)

31..24	23..18	17..12	11..6	5..0
S.MINOR	rd	rc	rb	op
8	6	6	6	6

[22] S.8 need not specify byte ordering, nor need it specify alignment checking, as it stores a single byte.

Description

An operand size, expressed in bytes, is specified by the instruction. A virtual address is computed from the sum of the contents of register rc and the contents of register rb multiplied by operand size. The contents of register rd, treated as the size specified, is stored in memory using the specified byte order.

If alignment is specified, the computed virtual address must be aligned, that is, it must be an exact multiple of the size expressed in bytes. If the address is not aligned an "access disallowed by virtual address" exception occurs.

Definition

```
def Store(op,rd,rc,rb) as
    case op of
        S8:
            size ← 8
        S16L, S16AL, S16B, S16AB:
            size ← 16
        S32L, S32AL, S32B, S32AB:
            size ← 32
        S64L, S64AL, S64B, S64AB,
        SMUX64AB, SMUX64AL:
            size ← 64
        S128L, S128AL, S128B, S128AB:
            size ← 128
    endcase
    lsize ← log(size)
    case op of
        S8:
            order ← undefined
        S16L, S32L, S64L, S128L,
        S16AL, S32AL, S64AL, S128AL, SMUX64AL:
            order ← L
        S16B, S32B, S64B, S128B,
        S16AB, S32AB, S64AB, S128AB, SMUX64AB:
            order ← B
    endcase
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 64)
    VirtAddr ← c + (b66-lsize..0 || 0^(lsize-3))
    case op of
        S16AL, S32AL, S64AL, S128AL,
        S16AB, S32AB, S64AB, S128AB,
        SMUX64AB, SMUX64AL:
            if (clsize-4..0 ≠ 0 then
                raise AccessDisallowedByVirtualAddress
            endif
        S16L, S32L, S64L, S128L,
        S16B, S32B, S64B, S128B:
        S8:
    endcase
    d ← RegRead(rd, 128)
    case op of
```

```
S8,
S16L, S16AL, S16B, S16AB,
S32L, S32AL, S32B, S32AB,
S64L, S64AL, S64B, S64AB,
S128L, S128AL, S128B, S128AB:
        StoreMemory(c,VirtAddr,size,order,d[size-1..0])
SMUX64AB, SMUX64AL:
        lock
                a ← LoadMemoryW(c,VirtAddr,size,order)
                m ← (d[127..64] & d[63..0]) | (a & ~d[63..0])
                StoreMemory(c,VirtAddr,size,order,m)
        endlock
    endcase
enddef
```

Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

Store Double Compare Swap

These operations compare two 64-bit values in a register against two 64-bit values read from two 64-bit memory locations, as specified by two 64-bit addresses in a register, and if equal, store two new 64-bit values from a register into the memory locations. The values read from memory are catenated and placed in a register.

Operation codes

S.D.C.S.64.AB	Store double compare swap octlet aligned big-endian
S.D.C.S.64.AL	Store double compare swap octlet aligned little-endian

Format

op rd@rc,rb

rd=op(rd,rc,rb)

31..24	23..18	17..12	11..6	5..0
S.MINOR	rd	rc	rb	op
8	6	6	6	6

Description

Two virtual addresses are extracted from the low order bits of the contents of registers rc and rb. Two 64-bit comparison values are extracted from the high order bits of the contents of registers rc and rb. Two 64-bit replacement values are extracted from the contents of register rd. The contents of memory using the specified byte order are read from the specified addresses, treated as 64-bit values, compared against the specified comparison values, and if both read values are equal to the comparison values, the two replacement values are written to memory using the specified byte order. If either are unequal, no values are written to memory. The loaded values are catenated and placed in the register specified by rd.

The virtual addresses must be aligned, that is, it must be an exact multiple of the size expressed in bytes. If the address is not aligned an "access disallowed by virtual address" exception occurs.

Definition

```
def StoreDoubleCompareSwap(op,rd,rc,rb) as
    size ← 64
    lsize ← log(size)
    case op of
        SDCS64AL:
            order ← l
        SDCS64AB:
```

```
            order ← 8
    endcase
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    d ← RegRead(rd, 128)
    if (c2..0 ≠ 0) or (b2..0 ≠ 0) then
        raise AccessDisallowedByVirtualAddress
    endif
    lock
        a ← LoadMemoryW(c63..0,c63..0,64,order) || LoadMemoryW(b63..0,b63..0,64,order)
        if ((c127..64 || b127..64) = a) then
            StoreMemory(c63..0,c63..0,64,order,d127..64)
            StoreMemory(b63..0,b63..0,64,order,d63..0)
        endif
    endlock
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

Store Immediate

These operations add the contents of a register to a sign-extended immediate value to produce a virtual address, and store the contents of a register into memory.

Operation codes

S.I.8[23]	Store immediate byte
S.I.16AB	Store immediate double aligned big-endian
S.I.16.B	Store immediate double big-endian
S.I.16AL	Store immediate double aligned little-endian
S.I.16.L	Store immediate double little-endian
S.I.32AB	Store immediate quadlet aligned big-endian
S.I.32.B	Store immediate quadlet big-endian
S.I.32AL	Store immediate quadlet aligned little-endian
S.I.32.L	Store immediate quadlet little-endian
S.I.64AB	Store immediate octlet aligned big-endian
S.I.64.B	Store immediate octlet big-endian
S.I.64AL	Store immediate octlet aligned little-endian
S.I.64.L	Store immediate octlet little-endian
S.I.128AB	Store immediate hexlet aligned big-endian
S.I.128.B	Store immediate hexlet big-endian
S.I.128AL	Store immediate hexlet aligned little-endian
S.I.128.L	Store immediate hexlet little-endian
S.MUXI.64AB	Store multiplex immediate octlet aligned big-endian
S.MUXI.64AL	Store multiplex immediate octlet aligned little-endian

Selection

number format	op	size	alignment	ordering
byte		8		
integer		16 32 64 128		L B
integer aligned		16 32 64 128	A	L B
multiplex	MUX	64	A	L B

Format

S.op.I.size.align.order rd,rc,offset

sopisizealignorder(rd,rc,offset)

31..24	23..18	17..12	11..0
op	rd	rc	offset
8	6	6	12

[23] S.I.8 need not specify byte ordering, nor need it specify alignment checking, as it stores a single byte.

Description

An operand size, expressed in bytes, is specified by the instruction. A virtual address is computed from the sum of the contents of register rc and the sign-extended value of the offset field, multiplied by the operand size. The contents of register rd, treated as the size specified, are written to memory using the specified byte order.

The computed virtual address must be aligned, that is, it must be an exact multiple of the size expressed in bytes. If the address is not aligned an "access disallowed by virtual address" exception occurs.

Definition

```
def StoreImmediate(op,rd,rc,offset) as
    case op of
        SI8:
            size ← 8
        SI16L, SI16AL, SI16B, SI16AB:
            size ← 16
        SI32L, SI32AL, SI32B, SI32AB:
            size ← 32
        SI64L, SI64AL, SI64B, SI64AB, SMUXI64AB, SMUXI64AL:
            size ← 64
        SI128L, SI128AL, SI128B, SI128AB:
            size ← 128
    endcase
    lsize ← log(size)
    case op of
        SI8:
            order ← undefined
        SI16L, SI32L, SI64L, SI128L,
        SI16AL, SI32AL, SI64AL, SI128AL, SMUXI64AL:
            order ← L
        SI16B, SI32B, SI64B, SI128B,
        SI16AB, SI32AB, SI64AB, SI128AB, SMUXI64AB:
            order ← B
    endcase
    c ← RegRead(rc, 64)
    VirtAddr ← c + (offset_5)^(5-lsize) || offset || 0^(lsize-3))
    case op of
        SI16AL, SI32AL, SI64AL, SI128AL,
        SI16AB, SI32AB, SI64AB, SI128AB,
        SMUXI64AB, SMUXI64AL:
            if (c_(lsize-4..0) ≠ 0 then
                raise AccessDisallowedByVirtualAddress
            endif
        SI16L, SI32L, SI64L, SI128L,
        SI16B, SI32B, SI64B, SI128B:
        SI8:
    endcase
    d ← RegRead(rd, 128)
    case op of
        SI8,
        SI16L, SI16AL, SI16B, SI16AB,
        SI32L, SI32AL, SI32B, SI32AB,
```

```
        SI64L, SI64AL, SI64B, SI64AB,
        SI128L, SI128AL, SI128B, SI128AB:
            StoreMemory(c,VirtAddr,size,order,d_size-1..0)
        SMUX64AB, SMUX64AL:
            lock
                a ← LoadMemoryW(c,VirtAddr,size,order)
                m ← (d127..64 & d63..0) | (a & ~d63..0)
                StoreMemory(c,VirtAddr,size,order,m)
            endlock
    endcase
enddef
```

Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

Store Immediate Inplace

These operations add the contents of a register to a sign-extended immediate value to produce a virtual address, and store the contents of a register into memory.

Operation codes

S.AS.I.64.AB	Store add swap immediate octlet aligned big-endian
S.AS.I.64.AL	Store add swap immediate octlet aligned little-endian
S.CS.I.64.AB	Store compare swap immediate octlet aligned big-endian
S.CS.I.64.AL	Store compare swap immediate octlet aligned little-endian
S.MS.I.64.AB	Store multiplex swap immediate octlet aligned big-endian
S.MS.I.64.AL	Store multiplex swap immediate octlet aligned little-endian

Selection

number format	op	size	alignment	ordering	
add-swap	AS	64	A	L	B
compare-swap	CS	64	A	L	B
multiplex-swap	MS	64	A	L	B

Format

S.op.I.64.align.order	rd@rc,offset

rd=sopi64alignorder(rd,rc,offset)

31..24	23..18	17..12	11..0
op	rd	rc	offset
8	6	6	12

Description

A virtual address is computed from the sum of the contents of register rc and the sign extended value of the offset field. The contents of memory using the specified byte order are read and treated as a 64-bit value. A specified operation is performed between the memory contents and the original contents of register rd, and the result is written to memory using the specified byte order. The original memory contents are placed into register rd.

The computed virtual address must be aligned, that is, it must be an exact multiple of the size expressed in bytes. If the address is not aligned an "access disallowed by virtual address" exception occurs.

Definition

```
def StoreImmediateInplace(op,rd,rc,offset) as
    size ← 64
    lsize ← log(size)
    case op of
```

```
        SASI64AL, SCSI64AL, SMSI64AL:
            order ← L
        SASI64AB, SCSI64AB, SMSI64AB:
            order ← B
    endcase
    c ← RegRead(rc, 64)
    VirtAddr ← c + (offset)^(55-size) || offset || 0^(size-3))
    if (c_(size-4..0) ≠ 0 then
        raise AccessDisallowedByVirtualAddress
    endif
    d ← RegRead(rd, 128)
    case op of
        SASI64AB, SASI64AL:
            lock
                a ← LoadMemoryW(c,VirtAddr,size,order)
                StoreMemory(c,VirtAddr,size,order,d63..0+a)
            endlock
        SCSI64AB, SCSI64AL:
            lock
                a ← LoadMemoryW(c,VirtAddr,size,order)
                if (a = d63..0) then
                    StoreMemory(c,VirtAddr,size,order,d127..64)
                endif
            endlock
        SMSI64AB, SMSI64AL:
            lock
                a ← LoadMemoryW(c,VirtAddr,size,order)
                m ← (d127..64 & d63..0) | (a & ~d63..0)
                StoreMemory(c,VirtAddr,size,order,m)
            endlock
    endcase
    RegWrite(rd, 64, a)
enddef
```

Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

Store Inplace

These operations add the contents of two registers to produce a virtual address, and store the contents of a register into memory.

Operation codes

S.AS.64.AB	Store add swap octlet aligned big-endian
S.AS.64.AL	Store add swap octlet aligned little-endian
S.CS.64.AB	Store compare swap octlet aligned big-endian
S.CS.64.AL	Store compare swap octlet aligned little-endian
S.MS.64.AB	Store multiplex swap octlet aligned big-endian
S.MS.64.AL	Store multiplex swap octlet aligned little-endian

Selection

number format	op	size	alignment	ordering	
add-swap	AS	64	A	L	B
compare-swap	CS	64	A	L	B
multiplex-swap	MS	64	A	L	B

Format

op rd@rc,rb

rd=op(rd,rc,rb)

31..24	23..18	17..12	11..6	5..0
S.MINOR	rd	rc	rb	op
8	6	6	6	6

Description

A virtual address is computed from the sum of the contents of register rc and the contents of register rb multiplied by operand size. The contents of memory using the specified byte order are read and treated as 64 bits. A specified operation is performed between the memory contents and the original contents of register rd, and the result is written to memory using the specified byte order. The original memory contents are placed into register rd.

The computed virtual address must be aligned, that is, it must be an exact multiple of the size expressed in bytes. If the address is not aligned an "access disallowed by virtual address" exception occurs.

Definition

```
def StoreInplace(op,rd,rc,rb) as
    size ← 64
    lsize ← log(size)
    case op of
```

```
        SAS64AL, SCS64AL, SMS64AL:
            order ← L
        SAS64AB, SCS64AB, SMS64AB:
            order ← B
    endcase
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 64)
    VirtAddr ← c + (b66-size..0 || 0^(size-3))
    if csize-4..0 ≠ 0 then
        raise AccessDisallowedByVirtualAddress
    endif
    d ← RegRead(rd, 128)
    case op of
        SAS64AB, SAS64AL:
            lock
                a ← LoadMemoryW(c,VirtAddr,size,order)
                StoreMemory(c,VirtAddr,size,order,d63..0+a)
            endlock
        SCS64AB, SCS64AL:
            lock
                a ← LoadMemoryW(c,VirtAddr,size,order)
                if (a = d63..0) then
                    StoreMemory(c,VirtAddr,size,order,d127..64)
                endif
            endlock
        SMS64AB, SMS64AL:
            lock
                a ← LoadMemoryW(c,VirtAddr,size,order)
                m ← (d127..64 & d63..0) | (a & ~d63..0)
                StoreMemory(c,VirtAddr,size,order,m)
            endlock
    endcase
    RegWrite(rd, 64, a)
enddef
```

Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

Group Add

These operations take operands from two registers, perform operations on partitions of bits in the operands, and place the concatenated results in a third register.

Operation codes

G.ADD.8	Group add bytes
G.ADD.16	Group add doublets
G.ADD.32	Group add quadlets
G.ADD.64	Group add octlets
G.ADD.128	Group add hexlet
G.ADD.L.8	Group add limit signed bytes
G.ADD.L.16	Group add limit signed doublets
G.ADD.L.32	Group add limit signed quadlets
G.ADD.L.64	Group add limit signed octlets
G.ADD.L.128	Group add limit signed hexlet
G.ADD.LU.8	Group add limit unsigned bytes
G.ADD.LU.16	Group add limit unsigned doublets
G.ADD.LU.32	Group add limit unsigned quadlets
G.ADD.LU.64	Group add limit unsigned octlets
G.ADD.LU.128	Group add limit unsigned hexlet
G.ADD.8.O	Group add signed bytes check overflow
G.ADD.16.O	Group add signed doublets check overflow
G.ADD.32.O	Group add signed quadlets check overflow
G.ADD.64.O	Group add signed octlets check overflow
G.ADD.128.O	Group add signed hexlet check overflow
G.ADD.U.8.O	Group add unsigned bytes check overflow
G.ADD.U.16.O	Group add unsigned doublets check overflow
G.ADD.U.32.O	Group add unsigned quadlets check overflow
G.ADD.U.64.O	Group add unsigned octlets check overflow
G.ADD.U.128.O	Group add unsigned hexlet check overflow

Redundancies

G.ADD.size rd=rc,rc	⇔	G.SHL.I.size rd=rc,1
G.ADD.size.O rd=rc,rc	⇔	G.SHL.I.size.O rd=rc,1
G.ADD.U.size.O rd=rc,rc	⇔	G.SHL.I.U.size.O rd=rc,1

Format

G.op.size rd=rc,rb

rd=gopsize(rc,rb)

31 ... 24	23 ... 18	17 ... 12	11 ... 6	5 ... 0
G.size	rd	rc	rb	op
8	6	6	6	6

Description

The contents of registers rc and rb are partitioned into groups of operands of the size specified and added, and if specified, checked for overflow or limited, yielding a group of results, each of which is the size specified. The group of results is catenated and placed in register rd.

Definition

```
def Group(op,size,rd,rc,rb)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    case op of
        G.ADD:
            for i ← 0 to 128-size by size
                a[i+size-1..i] ← c[i+size-1..i] + b[i+size-1..i]
            endfor
        G.ADD.L:
            for i ← 0 to 128-size by size
                t ← (c[i+size-1] || c[i+size-1..i]) + (b[i+size-1] || b[i+size-1..i])
                a[i+size-1..i] ← (t[size] ≠ t[size-1]) ? (t[size] || t[size]^(size-1)) : t[size-1..0]
            endfor
        G.ADD.L.U:
            for i ← 0 to 128-size by size
                t ← (0^1 || c[i+size-1..i]) + (0^1 || b[i+size-1..i])
                a[i+size-1..i] ← (t[size] ≠ 0) ? (1^size) : t[size-1..0]
            endfor
        G.ADD.O:
            for i ← 0 to 128-size by size
                t ← (c[i+size-1] || c[i+size-1..i]) + (b[i+size-1] || b[i+size-1..i])
                if t[size] ≠ t[size-1] then
                    raise FixedPointArithmetic
                endif
                a[i+size-1..i] ← t[size-1..0]
            endfor
        G.ADD.U.O:
            for i ← 0 to 128-size by size
                t ← (0^1 || c[i+size-1..i]) + (0^1 || b[i+size-1..i])
                if t[size] ≠ 0 then
                    raise FixedPointArithmetic
                endif
                a[i+size-1..i] ← t[size-1..0]
```

```
            endfor
    endcase
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Fixed point arithmetic

Group Add Halve

These operations take operands from two registers, perform operations on partitions of bits in the operands, and place the concatenated results in a third register.

Operation codes

GADD.H.8.C	Group add halve signed bytes ceiling
GADD.H.8.F	Group add halve signed bytes floor
GADD.H.8.N	Group add halve signed bytes nearest
GADD.H.8.Z	Group add halve signed bytes zero
GADD.H.16.C	Group add halve signed doublets ceiling
GADD.H.16.F	Group add halve signed doublets floor
GADD.H.16.N	Group add halve signed doublets nearest
GADD.H.16.Z	Group add halve signed doublets zero
GADD.H.32.C	Group add halve signed quadlets ceiling
GADD.H.32.F	Group add halve signed quadlets floor
GADD.H.32.N	Group add halve signed quadlets nearest
GADD.H.32.Z	Group add halve signed quadlets zero
GADD.H.64.C	Group add halve signed octlets ceiling
GADD.H.64.F	Group add halve signed octlets floor
GADD.H.64.N	Group add halve signed octlets nearest
GADD.H.64.Z	Group add halve signed octlets zero
GADD.H.128.C	Group add halve signed hexlet ceiling
GADD.H.128.F	Group add halve signed hexlet floor
GADD.H.128.N	Group add halve signed hexlet nearest
GADD.H.128.Z	Group add halve signed hexlet zero
GADD.H.U.8.C	Group add halve unsigned bytes ceiling
GADD.H.U.8.F	Group add halve unsigned bytes floor
GADD.H.U.8.N	Group add halve unsigned bytes nearest
GADD.H.U.16.C	Group add halve unsigned doublets ceiling
GADD.H.U.16.F	Group add halve unsigned doublets floor
GADD.H.U.16.N	Group add halve unsigned doublets nearest
GADD.H.U.32.C	Group add halve unsigned quadlets ceiling
GADD.H.U.32.F	Group add halve unsigned quadlets floor
GADD.H.U.32.N	Group add halve unsigned quadlets nearest
GADD.H.U.64.C	Group add halve unsigned octlets ceiling
GADD.H.U.64.F	Group add halve unsigned octlets floor
GADD.H.U.64.N	Group add halve unsigned octlets nearest
GADD.H.U.128.C	Group add halve unsigned hexlet ceiling
GADD.H.U.128.F	Group add halve unsigned hexlet floor
GADD.H.U.128.N	Group add halve unsigned hexlet nearest

Redundancies

G.ADD.H.size.rnd rd=rc,rc	⇔	G.COPY rd=rc
G.ADD.H.U.size.rnd rd=rc,rc	⇔	G.COPY rd=rc

Format

G.op.size.rnd rd=rc,rb

rd=gopsizernd(rc,rb)

31..24	23..18	17..12	11..6	5..2	1..0
G.size	rd	rc	rb	op	rnd
8	6	6	6	4	2

Description

The contents of registers rc and rb are partitioned into groups of operands of the size specified, added, halved, and rounded as specified, yielding a group of results, each of which is the size specified. The results never overflow, so limiting is not required by this operation. The group of results is catenated and placed in register rd.

Z (zero) rounding is not defined for unsigned operations, and a ReservedInstruction exception is raised if attempted. F (floor) rounding will properly round unsigned results downward.

Definition

```
def GroupAddHalve(op,rnd,size,rd,rc,rb)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    case op of
        GADDHC, GADDHF, GADDHN, GADDHZ:
            as ← cs ← bs ← 1
        GADDHUC, GADDHUF, GADDHUN, GADDHUZ
            as ← cs ← bs ← 0
            if rnd = Z then
                raise ReservedInstruction
            endif
    endcase
    h ← size+1
    r ← 1
    for i ← 0 to 128-size by size
        p ← ((cs and c_{size-1}) || c_{size-1+i..i}) + ((bs and b_{size-1}) || b_{size-1+i..i})
        case rnd of
            none, N:
                s ← 0^{size} || -p_1
            Z:
                s ← 0^{size} || p_{size}
            F:
                s ← 0^{size+1}
            C:
```

```
            s ← 0^size || 1^1
        endcase
        v ← ((as & p_size)||p) + (0||s)
        a_size-1+i..i ← v_size..1
    endfor
    RegWrite(rd, 128, a)
enddef
```

Exceptions

ReservedInstruction

Group Boolean

These operations take operands from three registers, perform boolean operations on corresponding bits in the operands, and place the concatenated results in the third register.

Operation codes

G.BOOLEAN	Group boolean

Equivalencies

G.AAA	Group three-way and
G.AAA.1	Group add add add bits
G.AAS.1	Group add add subtract bits
G.ADD.1	Group add bits
G.AND	Group and
G.ANDN	Group and not
G.COPY	Group copy
G.NAAA	Group three-way nand
G.NAND	Group nand
G.NOOO	Group three-way nor
G.NOR	Group nor
G.NOT	Group not
G.NXXX	Group three-way exclusive-nor
G.OOO	Group three-way or
G.OR	Group or
G.ORN	Group or not
G.SAA.1	Group subtract add add bits
G.SAS.1	Group subtract add subtract bits
G.SET	Group set
G.SET.AND.E.1	Group set and equal zero bits
G.SET.AND.NE.1	Group set and not equal zero bits
G.SET.E.1	Group set equal bits
G.SET.G.1	Group set greater signed bits
G.SET.G.U.1	Group set greater unsigned bits
G.SET.G.Z.1	Group set greater zero signed bits
G.SET.GE.1	Group set greater equal signed bits
G.SET.GE.Z.1	Group set greater equal zero signed bits
G.SET.L.1	Group set less signed bits
G.SET.L.Z.1	Group set less zero signed bits
G.SET.LE.1	Group set less equal signed bits
G.SET.LE.U.1	Group set less equal unsigned bits
G.SET.LE.Z.1	Group set less equal zero signed bits
G.SET.NE.1	Group set not equal bits
G.SET.GE.U.1	Group set greater equal unsigned bits
G.SET.L.U.1	Group set less unsigned bits

G.SSA.1	Group subtract subtract add bits
G.SSS.1	Group subtract subtract subtract bits
G.SUB.1	Group subtract bits
G.XNOR	Group exclusive-nor
G.XOR	Group exclusive-or
G.XXX	Group three-way exclusive-or
G.ZERO	Group zero

G.AAA rd@rc,rb	←	G.BOOLEAN rd@rc,rb,0b10000000
G.AAA.1 rd@rc,rb	→	G.XXX rd@rc,rb
G.AAS.1 rd@rc,rb	→	G.XXX rd@rc,rb
G.ADD.1 rd=rc,rb	→	G.XOR rd=rc,rb
G.AND rd=rc,rb	←	G.BOOLEAN rd@rc,rb,0b10001000
G.ANDN rd=rc,rb	←	G.BOOLEAN rd@rc,rb,0b01000100
G.BOOLEAN rd@rb,rc,i	→	G.BOOLEAN rd@rc,rb,$i_7i_5i_6i_4i_3i_1i_2i_0$
G.COPY rd=rc	←	G.BOOLEAN rd@rc,rc,0b10001000
G.NAAA rd@rc,rb	←	G.BOOLEAN rd@rc,rb,0b01111111
G.NAND rd=rc,rb	←	G.BOOLEAN rd@rc,rb,0b01110111
G.NOOO rd@rc,rb	←	G.BOOLEAN rd@rc,rb,0b00000001
G.NOR rd=rc,rb	←	G.BOOLEAN rd@rc,rb,0b00010001
G.NOT rd=rc	←	G.BOOLEAN rd@rc,rc,0b00010001
G.NXXX rd@rc,rb	←	G.BOOLEAN rd@rc,rb,0b01101001
G.OOO rd@rc,rb	←	G.BOOLEAN rd@rc,rb,0b11111110
G.OR rd=rc,rb	←	G.BOOLEAN rd@rc,rb,0b11101110
G.ORN rd=rc,rb	←	G.BOOLEAN rd@rc,rb,0b11011101
G.SAA.1 rd@rc,rb	→	G.XXX rd@rc,rb
G.SAS.1 rd@rc,rb	→	G.XXX rd@rc,rb
G.SET rd	←	G.BOOLEAN rd@rd,rd,0b10000001
G.SET.AND.E.1 rd=rb,rc	→	G.NAND rd=rc,rb
G.SET.AND.NE.1 rd=rb,rc	→	G.AND rd=rc,rb
G.SET.E.1 rd=rb,rc	→	G.XNOR rd=rc,rb
G.SET.G.1 rd=rb,rc	→	G.ANDN rd=rc,rb
G.SET.G.U.1 rd=rb,rc	→	G.ANDN rd=rb,rc
G.SET.G.Z.1 rd=rc	→	G.ZERO rd
G.SET.GE.1 rd=rb,rc	→	G.ORN rd=rc,rb
G.SET.GE.Z.1 rd=rc	→	G.NOT rd=rc
G.SET.L.1 rd=rb,rc	→	G.ANDN rd=rb,rc
G.SET.L.Z.1 rd=rc	→	G.COPY rd=rc
G.SET.LE.1 rd=rb,rc	→	G.ORN rd=rb,rc
G.SET.LE.U.1 rd=rb,rc	→	G.ORN rd=rc,rb
G.SET.LE.Z.1 rd=rc	→	G.SET rd
G.SET.NE.1 rd=rb,rc	→	G.XOR rd=rc,rb

G.SET.GE.U.1 rd=rb,rc	→	G.ORN rd=rb,rc
G.SET.L.U.1 rd=rb,rc	→	G.ANDN rd=rc,rb
G.SSA.1 rd@rc,rb	→	G.XXX rd@rc,rb
G.SSS.1 rd@rc,rb	→	G.XXX rd@rc,rb
G.SUB.1 rd=rc,rb	→	G.XOR rd=rc,rb
G.XNOR rd=rc,rb	←	G.BOOLEAN rd@rc,rb,0b10011001
G.XOR rd=rc,rb	←	G.BOOLEAN rd@rc,rb,0b01100110
G.XXX rd@rc,rb	←	G.BOOLEAN rd@rc,rb,0b10010110
G.ZERO rd	←	G.BOOLEAN rd@rd,rd,0b00000000

Selection

operation	function (binary)	function (decimal)
d	11110000	240
c	11001100	204
b	10101010	176
d&c&b	10000000	128
(d&c)|b	11101010	234
d|c|b	11111110	254
d?c:b	11001010	202
d^c^b	10010110	150
~d^c^b	01101001	105
0	00000000	0

Format

G.BOOLEAN rd@trc,trb,f

rd=gbooleani(rd,rc,rb,f)

31–25	24	23–18	17–12	11–6	5–0
G.BOOLEAN	ih	rd	rc	rb	il
7	1	6	6	6	6

```
if f6=f5 then
    if f2=f1 then
        if f2 then
            rc ← max(trc,trb)
            rb ← min(trc,trb)
        else
            rc ← min(trc,trb)
            rb ← max(trc,trb)
        endif
        ih ← 0
        il ← 0 || f6 || f7 || f4 || f3 || f0
    else
        if f2 then
            rc ← trb
            rb ← trc
        else
            rc ← trc
            rb ← trb
        endif
        ih ← 0
        il ← 1 || f6 || f7 || f4 || f3 || f0
    endif
else
    ih ← 1
    if f6 then
        rc ← trb
        rb ← trc
        il ← f1 || f2 || f7 || f4 || f3 || f0
    else
        rc ← trc
        rb ← trb
        il ← f2 || f1 || f7 || f4 || f3 || f0
    endif
endif
```

Description

Three values are taken from the contents of registers rd, rc and rb. The ih and il fields specify a function of three bits, producing a single bit result. The specified function is evaluated for each bit position, and the results are catenated and placed in register rd.

Register rd is both a source and destination of this instruction.

The function is specified by eight bits, which give the result for each possible value of the three source bits in each bit position:

d	1 1 1 1 0 0 0 0
c	1 1 0 0 1 1 0 0
b	1 0 1 0 1 0 1 0
f(d,c,b)	$f_7\ f_6\ f_5\ f_4\ f_3\ f_2\ f_1\ f_0$

A function can be modified by rearranging the bits of the immediate value. The table below shows how rearrangement of immediate value $f_{7..0}$ can reorder the operands d,c,b for the same function.

operation	immediate
f(d,c,b)	$f_7\ f_6\ f_5\ f_4\ f_3\ f_2\ f_1\ f_0$
f(c,d,b)	$f_7\ f_6\ f_3\ f_2\ f_5\ f_4\ f_1\ f_0$
f(d,b,c)	$f_7\ f_5\ f_6\ f_4\ f_3\ f_1\ f_2\ f_0$
f(b,c,d)	$f_7\ f_3\ f_5\ f_1\ f_6\ f_2\ f_4\ f_0$
f(c,b,d)	$f_7\ f_5\ f_3\ f_1\ f_6\ f_4\ f_2\ f_0$
f(b,d,c)	$f_7\ f_3\ f_6\ f_2\ f_5\ f_1\ f_4\ f_0$

By using such a rearrangement, an operation of the form: *b=f(d,c,b)* can be recoded into a legal form: *b=f(b,d,c)*. For example, the function: *b=f(d,c,b)=d?c:b* cannot be coded, but the equivalent function: *d=c?b:d* can be determined by rearranging the code for *d= f(d,c,b)=d?c:b*, which is 11001010, according to the rule for *f(d,c,b)⇒f(c,b,d)*, to the code 11011000.

Encoding

Some special characteristics of this rearrangement is the basis of the manner in which the eight function specification bits are compressed to seven immediate bits in this instruction. As seen in the table above, in the general case, a rearrangement of operands from *f(d,c,b)* to *f(d,b,c)*.(interchanging rc and rb) requires interchanging the values of f_6 and f_5 and the values of f_2 and f_1.

Among the 256 possible functions which this instruction can perform, one quarter of them (64 functions) are unchanged by this rearrangement. These functions have the property that $f_6=f_5$ and $f_2=f_1$. The values of rc and rb[24] can be freely interchanged, and so are sorted into rising or falling order to indicate the value of f_2.[25] These functions are encoded by the values of f_7, f_6, f_4, f_3, and f_0 in the immediate field and f_2 by whether rc>rb, thus using 32 immediate values for 64 functions.

Another quarter of the functions have $f_6=1$ and $f_5=0$. These functions are recoded by interchanging rc and rb, f_6 and f_5, f_2 and f_1. They then share the same encoding as the

---

[24] Note that rc and rb are the register specifiers, not the register contents.

[25] A special case arises when rc=rb, so the sorting of rc and rb cannot convey information. However, as only the values f_7, f_4, f_3, and f_0 can ever result in this case, f_6, f_5, f_2, and f_1 need not be coded for this case, so no special handling is required.

quarter of the functions where $f_6=0$ and $f_5=1$, and are encoded by the values of f_7, f_4, f_3, f_2, f_1, and f_0 in the immediate field, thus using 64 immediate values for 128 functions.

The remaining quarter of the functions have $f_6=f_5$ and $f_2 \neq f_1$. The half of these in which $f_2=1$ and $f_1=0$ are recoded by interchanging rc and rb, f_6 and f_5, f_2 and f_1. They then share the same encoding as the eighth of the functions where $f_2=0$ and $f_1=1$, and are encoded by the values of f_7, f_6, f_4, f_3, and f_0 in the immediate field, thus using 32 immediate values for 64 functions.

The function encoding is summarized by the table:

f_7	f_6	f_5	f_4	f_3	f_2	f_1	f_0	trc>trb	ih	il_5	il_4	il_3	il_2	il_1	il_0	rc	rb
		f_6				f_2		f_2	0	0	f_6	f_7	f_4	f_3	f_0	trc	trb
		f_6				f_2		$\sim f_2$	0	0	f_6	f_7	f_4	f_3	f_0	trb	trc
		f_6			0	1			0	1	f_6	f_7	f_4	f_3	f_0	trc	trb
		f_6			1	0			0	1	f_6	f_7	f_4	f_3	f_0	trb	trc
	0	1							1	f_2	f_1	f_7	f_4	f_3	f_0	trc	trb
	1	0							1	f_1	f_2	f_7	f_4	f_3	f_0	trb	trc

The function decoding is summarized by the table:

ih	il_5	il_4	il_3	il_2	il_1	il_0	rc>rb	f_7	f_6	f_5	f_4	f_3	f_2	f_1	f_0
0	0						0	il_3	il_4	il_4	il_2	il_1	0	0	il_0
0	0						1	il_3	il_4	il_4	il_2	il_1	1	1	il_0
0	1							il_3	il_4	il_4	il_2	il_1	0	1	il_0
1								il_3	0	1	il_2	il_1	il_5	il_4	il_0

Definition

```
def GroupBoolean (ih,rd,rc,rb,il)
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    if ih=0 then
        if il5=0 then
            f ← il3 || il4 || il4 || il2 || il1 || (rc>rb)^2 || il0
        else
            f ← il3 || il4 || il4 || il2 || il1 || 0 || 1 || il0
        endif
    else
        f ← il3 || 0 || 1 || il2 || il1 || il5 || il4 || il0
    endif
    for i ← 0 to 127 by size
        ai ← f[di||ci||bi]
    endfor
    RegWrite(rd, 128, a)
enddef
```

Exceptions

none

Group Compare

These operations perform calculations on partitions of bits in two general register values, and generate a fixed-point arithmetic exception if the condition specified is met.

Operation codes

G.COM.AND.E.8	Group compare and equal zero bytes
G.COM.AND.E.16	Group compare and equal zero doublets
G.COM.AND.E.32	Group compare and equal zero quadlets
G.COM.AND.E.64	Group compare and equal zero octlets
G.COM.AND.E.128	Group compare and equal zero hexlet
G.COM.AND.NE.8	Group compare and not equal zero bytes
G.COM.AND.NE.16	Group compare and not equal zero doublets
G.COM.AND.NE.32	Group compare and not equal zero quadlets
G.COM.AND.NE.64	Group compare and not equal zero octlets
G.COM.AND.NE.128	Group compare and not equal zero hexlet
G.COM.E.8	Group compare equal bytes
G.COM.E.16	Group compare equal doublets
G.COM.E.32	Group compare equal quadlets
G.COM.E.64	Group compare equal octlets
G.COM.E.128	Group compare equal hexlet
G.COM.GE.8	Group compare greater equal signed bytes
G.COM.GE.16	Group compare greater equal signed doublets
G.COM.GE.32	Group compare greater equal signed quadlets
G.COM.GE.64	Group compare greater equal signed octlets
G.COM.GE.128	Group compare greater equal signed hexlet
G.COM.GE.U.8	Group compare greater equal unsigned bytes
G.COM.GE.U.16	Group compare greater equal unsigned doublets
G.COM.GE.U.32	Group compare greater equal unsigned quadlets
G.COM.GE.U.64	Group compare greater equal unsigned octlets
G.COM.GE.U.128	Group compare greater equal unsigned hexlet
G.COM.L.8	Group compare signed less bytes
G.COM.L.16	Group compare signed less doublets
G.COM.L.32	Group compare signed less quadlets
G.COM.L.64	Group compare signed less octlets
G.COM.L.128	Group compare signed less hexlet
G.COM.L.U.8	Group compare less unsigned bytes
G.COM.L.U.16	Group compare less unsigned doublets
G.COM.L.U.32	Group compare less unsigned quadlets
G.COM.L.U.64	Group compare less unsigned octlets
G.COM.L.U.128	Group compare less unsigned hexlet
G.COM.NE.8	Group compare not equal bytes
G.COM.NE.16	Group compare not equal doublets
G.COM.NE.32	Group compare not equal quadlets
G.COM.NE.64	Group compare not equal octlets

G.COM.NE 128	Group compare not equal hexlet

Equivalencies

G.COM.E.Z.8	Group compare equal zero signed bytes
G.COM.E.Z.16	Group compare equal zero signed doublets
G.COM.E.Z.32	Group compare equal zero signed quadlets
G.COM.E.Z.64	Group compare equal zero signed octlets
G.COM.E.Z.128	Group compare equal zero signed hexlet
G.COM.G.8	Group compare signed greater bytes
G.COM.G.16	Group compare signed greater doublets
G.COM.G.32	Group compare signed greater quadlets
G.COM.G.64	Group compare signed greater octlets
G.COM.G.128	Group compare signed greater hexlet
G.COM.G.U.8	Group compare greater unsigned bytes
G.COM.G.U.16	Group compare greater unsigned doublets
G.COM.G.U.32	Group compare greater unsigned quadlets
G.COM.G.U.64	Group compare greater unsigned octlets
G.COM.G.U.128	Group compare greater unsigned hexlet
G.COM.G.Z.8	Group compare greater zero signed bytes
G.COM.G.Z.16	Group compare greater zero signed doublets
G.COM.G.Z.32	Group compare greater zero signed quadlets
G.COM.G.Z.64	Group compare greater zero signed octlets
G.COM.G.Z.128	Group compare greater zero signed hexlet
G.COM.GE.Z.8	Group compare greater equal zero signed bytes
G.COM.GE.Z.16	Group compare greater equal zero signed doublets
G.COM.GE.Z.32	Group compare greater equal zero signed quadlets
G.COM.GE.Z.64	Group compare greater equal zero signed octlets
G.COM.GE.Z.128	Group compare greater equal zero signed hexlet
G.COM.L.Z.8	Group compare less zero signed bytes
G.COM.L.Z.16	Group compare less zero signed doublets
G.COM.L.Z.32	Group compare less zero signed quadlets
G.COM.L.Z.64	Group compare less zero signed octlets
G.COM.L.Z.128	Group compare less zero signed hexlet
G.COM.LE.8	Group compare less equal signed bytes
G.COM.LE.16	Group compare less equal signed doublets
G.COM.LE.32	Group compare less equal signed quadlets
G.COM.LE.64	Group compare less equal signed octlets
G.COM.LE.128	Group compare less equal signed hexlet
G.COM.LE.U.8	Group compare less equal unsigned bytes
G.COM.LE.U.16	Group compare less equal unsigned doublets
G.COM.LE.U.32	Group compare less equal unsigned quadlets
G.COM.LE.U.64	Group compare less equal unsigned octlets
G.COM.LE.U.128	Group compare less equal unsigned hexlet
G.COM.LE.Z.8	Group compare less equal zero signed bytes
G.COM.LE.Z.16	Group compare less equal zero signed doublets
G.COM.LE.Z.32	Group compare less equal zero signed quadlets
G.COM.LE.Z.64	Group compare less equal zero signed octlets

G.COM.LE.Z.128	Group compare less equal zero signed hexlet
G.COM.NE.Z.8	Group compare not equal zero signed bytes
G.COM.NE.Z.16	Group compare not equal zero signed doublets
G.COM.NE.Z.32	Group compare not equal zero signed quadlets
G.COM.NE.Z.64	Group compare not equal zero signed octlets
G.COM.NE.Z.128	Group compare not equal zero signed hexlet
G.FIX	Group fixed point arithmetic exception
G.NOP	Group no operation

G.COM.E.Z.size rc	←	G.COM.AND.E.size rc,rc
G.COM.G.size rd,rc	→	G.COM.L.size rc,rd
G.COM.G.U.size rd,rc	→	G.COM.L.U.size rc,rd
G.COM.G.Z.size rc	⇐	G.COM.L.U.size rc,rc
G.COM.GE.Z.size rc	⇐	G.COM.GE.size rc,rc
G.COM.L.Z.size rc	⇐	G.COM.L.size rc,rc
G.COM.LE.size rd,rc	→	G.COM.GE.size rc,rd
G.COM.LE.U.size rd,rc	→	G.COM.GE.U.size rc,rd
G.COM.LE.Z.size rc	⇐	G.COM.GE.U.size rc,rc
G.COM.NE.Z.size rc	←	G.COM.AND.NE.size rc,rc
G.FIX	←	G.COM.E.128 r0,r0
G.NOP	←	G.COM.NE.128 r0,r0

Redundancies

G.COM.E.size rd,rd	⇔	*G.FIX*
G.COM.NE.size rd,rd	⇔	*G.NOP*

Selection

class	operation	cond	type	size
boolean	COM.AND COM	E NE		8 16 32 64 128
arithmetic	COM	L GE *G LE*	NONE U	8 16 32 64 128
	COM	*L GE G LE E NE*	*Z*	8 16 32 64 128

Format

G.COM.op.size rd,rc
G.COM.opz.size rcd

gcomopsize(rd,rc)

31..24	23..18	17..12	11..6	5..0
G.size	rd	rc	op	GCOM
8	6	6	6	6

Description

Two values are taken from the contents of registers rd and rc. The specified condition is calculated on partitions of the operands. If the specified condition is true for any partition, a fixed-point arithmetic exception is generated. This instruction generates no general purpose register results..

Definition

```
def GroupCompare(op,size,rd,rc)
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    case op of
        G.COM.E:
            for i ← 0 to 128-size by size
                a_{i+size-1..i} ← (d_{i+size-1..i} = c_{i+size-1..i})^size
            endfor
        G.COM.NE:
            for i ← 0 to 128-size by size
                a_{i+size-1..i} ← (d_{i+size-1..i} ≠ c_{i+size-1..i})^size
            endfor
        G.COM.AND.E:
            for i ← 0 to 128-size by size
                a_{i+size-1..i} ← ((c_{i+size-1..i} and d_{i+size-1..i}) = 0)^size
            endfor
        G.COM.AND.NE:
            for i ← 0 to 128-size by size
                a_{i+size-1..i} ← ((c_{i+size-1..i} and d_{i+size-1..i}) ≠ 0)^size
            endfor
        G.COM.L:
            for i ← 0 to 128-size by size
                a_{i+size-1..i} ← ((rd = rc) ? (c_{i+size-1..i} < 0) : (d_{i+size-1..i} < c_{i+size-1..i}))^size
            endfor
        G.COM.GE:
            for i ← 0 to 128-size by size
                a_{i+size-1..i} ← ((rd = rc) ? (c_{i+size-1..i} ≥ 0) : (d_{i+size-1..i} ≥ c_{i+size-1..i}))^size
            endfor
        G.COM.L.U:
            for i ← 0 to 128-size by size
                a_{i+size-1..i} ← ((rd = rc) ? (c_{i+size-1..i} > 0) :
                    ((0 || d_{i+size-1..i}) < (0 || c_{i+size-1..i})))^size
            endfor
        G.COM.GE.U:
            for i ← 0 to 128-size by size
                a_{i+size-1..i} ← ((rd = rc) ? (c_{i+size-1..i} ≤ 0) :
                    ((0 || d_{i+size-1..i}) ≥ (0 || c_{i+size-1..i})))^size
            endfor
    endcase
    if (a ≠ 0) then
        raise FixedPointArithmetic
    endif
enddef
```

Exceptions

Fixed-point arithmetic

Group Compare Floating-point

These operations perform calculations on partitions of bits in two general register values, and generate a floating-point arithmetic exception if the condition specified is met.

Operation codes

G.COM.E.F.16	Group compare equal floating-point half
G.COM.E.F.16.X	Group compare equal floating-point half exact
G.COM.E.F.32	Group compare equal floating-point single
G.COM.E.F.32.X	Group compare equal floating-point single exact
G.COM.E.F.64	Group compare equal floating-point double
G.COM.E.F.64.X	Group compare equal floating-point double exact
G.COM.E.F.128	Group compare equal floating-point quad
G.COM.E.F.128.X	Group compare equal floating-point quad exact
G.COM.GE.F.16	Group compare greater or equal floating-point half
G.COM.GE.F.16.X	Group compare greater or equal floating-point half exact
G.COM.GE.F.32	Group compare greater or equal floating-point single
G.COM.GE.F.32.X	Group compare greater or equal floating-point single exact
G.COM.GE.F.64	Group compare greater or equal floating-point double
G.COM.GE.F.64.X	Group compare greater or equal floating-point double exact
G.COM.GE.F.128	Group compare greater or equal floating-point quad
G.COM.GE.F.128.X	Group compare greater or equal floating-point quad exact
G.COM.L.F.16	Group compare less floating-point half
G.COM.L.F.16.X	Group compare less floating-point half exact
G.COM.L.F.32	Group compare less floating-point single
G.COM.L.F.32.X	Group compare less floating-point single exact
G.COM.L.F.64	Group compare less floating-point double
G.COM.L.F.64.X	Group compare less floating-point double exact
G.COM.L.F.128	Group compare less floating-point quad
G.COM.L.F.128.X	Group compare less floating-point quad exact
G.COM.LG.F.16	Group compare less or greater floating-point half
G.COM.LG.F.16.X	Group compare less or greater floating-point half exact
G.COM.LG.F.32	Group compare less or greater floating-point single
G.COM.LG.F.32.X	Group compare less or greater floating-point single exact
G.COM.LG.F.64	Group compare less or greater floating-point double
G.COM.LG.F.64.X	Group compare less or greater floating-point double exact
G.COM.LG.F.128	Group compare less or greater floating-point quad
G.COM.LG.F.128.X	Group compare less or greater floating-point quad exact

Equivalencies

G.COM.G.F.16	Group compare greater floating-point half
G.COM.G.F.16.X	Group compare greater floating-point half exact
G.COM.G.F.32	Group compare greater floating-point single
G.COM.G.F.32.X	Group compare greater floating-point single exact
G.COM.G.F.64	Group compare greater floating-point double
G.COM.G.F.64.X	Group compare greater floating-point double exact
G.COM.G.F.128	Group compare greater floating-point quad
G.COM.G.F.128.X	Group compare greater floating-point quad exact
G.COM.LE.F.16	Group compare less equal floating-point half
G.COM.LE.F.16.X	Group compare less equal floating-point half exact
G.COM.LE.F.32	Group compare less equal floating-point single
G.COM.LE.F.32.X	Group compare less equal floating-point single exact
G.COM.LE.F.64	Group compare less equal floating-point double
G.COM.LE.F.64.X	Group compare less equal floating-point double exact
G.COM.LE.F.128	Group compare less equal floating-point quad
G.COM.LE.F.128.X	Group compare less equal floating-point quad exact

G.COM.G.F.prec rd,rc	→	G.COM.L.F.prec rc,rd
G.COM.G.F.prec.X rd,rc	→	G.COM.L.F.prec.X rc,rd
G.COM.LE.F.prec rd,rc	→	G.COM.GE.F.prec rc,rd
G.COM.LE.F.prec.X rd,rc	→	G.COM.GE.F.prec.X rc,rd

Selection

class	op	cond	type	prec	round/trap
set	COM	E LG L GE G LE	F	16 32 64 128	NONE X

Format

G.COM.op.prec.round rd,rc

rc=gcomopprecround(rd,rc)

31..24	23..18	17..12	11..6	5..0
G.prec	rd	rc	op	GCOM
8	6	6	6	6

Description

The contents of registers rd and rc are compared using the specified floating-point condition. If the result of the comparison is true for any corresponding pair of elements, a floating point exception is raised. If a rounding option is specified, the operation raises a floating point exception if a floating point invalid operation occurs. If a rounding option is not specified, floating point exceptions are not raised, and are handled according to the default rules of IEEE 754.

Definition

```
def GroupCompareFloatingPoint(op,prec,round,rd,rc) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    for i ← 0 to 128-prec by prec
        di ← F(prec,d_{i+prec-1..i})
        ci ← F(prec,c_{i+prec-1..i})
        if round≠NONE then
            if (di.t = SNAN) or (ci.t = SNAN) then
                raise FloatingPointArithmetic
            endif
            case op of
                G.COM.L.F, G.COM.GE.F:
                    if (di.t = QNAN) or (ci.t = QNAN) then
                        raise FloatingPointArithmetic
                    endif
                others: //nothing
            endcase
        endif
        case op of
            G.COM.L.F:
                ai ← di?≥ci
            G.COM.GE.F:
                ai ← di!?<ci
            G.COM.E.F:
                ai ← di=ci
            G.COM.LG.F:
                ai ← di≠ci
        endcase
        a_{i+prec-1..i} ← ai
    endfor
    if (a ≠ 0) then
        raise FloatingPointArithmetic
    endif
enddef
```

Exceptions

Floating point arithmetic

Group Copy Immediate

This operation copies an immediate value to a general register.

Operation codes

G.COPY.I.16	Group copy immediate doublet
G.COPY.I.32	Group signed copy immediate quadlet
G.COPY.I.64	Group signed copy immediate octlet
G.COPY.I.128	Group signed copy immediate hexlet

Equivalencies

G.COPY.I.8	Group copy immediate byte
G.SET	Group set
G.ZERO	Group zero

G.COPY.I.8 rd=(i_7^8 || $i_{7..0}$)	←	G.COPY.I.16 rd=(0 || $i_{7..0}$ || $i_{7..0}$)
G.SET rd	←	G.COPY.I.128 rd=-1
G.ZERO rd	←	G.COPY.I.128 rd=0

Redundancies

G.COPY.I.size rd=-1	⇔	*G.SET rd*
G.COPY.I.size rd=0	⇔	*G.ZERO rd*

Format

G.COPY.I.size rd=i

rd=gcopyisize(i)

31..25	24	23..18	17..16	15..0
G.COPY.I	s	rd	size	imm
7	1	6	2	16

$s \leftarrow i_{16}$
$imm \leftarrow i_{15..0}$

Description

A 128-bit immediate value is produced from the operation code, the size field and the 16-bit imm field. The result is placed into register ra.

Definition

```
def GroupCopyImmediate(op,size,rd,imm) as
    s ← op0
```

```
    case size of
        16:
            if s then
                ReservedInstruction
            endif
            a ← imm || imm || imm || imm || imm || imm || imm || imm
        32:
            a ← s^16 || imm || s^16 || imm || s^16 || imm || s^16 || imm
        64:
            a ← s^48 || imm || s^48 || imm
        128:
            a ← s^112 || imm
    endcase
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Reserved Instruction

Group Immediate

These operations take operands from a register and an immediate value, perform operations on partitions of bits in the operands, and place the concatenated results in a second register.

Operation codes

G.ADD.I.16	Group add immediate doublet
G.ADD.I.16.O	Group add immediate signed doublet check overflow
G.ADD.I.32	Group add immediate quadlet
G.ADD.I.32.O	Group add immediate signed quadlet check overflow
G.ADD.I.64	Group add immediate octlet
G.ADD.I.64.O	Group add immediate signed octlet check overflow
G.ADD.I.128	Group add immediate hexlet
G.ADD.I.128.O	Group add immediate signed hexlet check overflow
G.ADD.I.U.16.O	Group add immediate unsigned doublet check overflow
G.ADD.I.U.32.O	Group add immediate unsigned quadlet check overflow
G.ADD.I.U.64.O	Group add immediate unsigned octlet check overflow
G.ADD.I.U.128.O	Group add immediate unsigned hexlet check overflow
G.AND.I.16	Group and immediate doublet
G.AND.I.32	Group and immediate quadlet
G.AND.I.64	Group and immediate octlet
G.AND.I.128	Group and immediate hexlet
G.NAND.I.16	Group not and immediate doublet
G.NAND.I.32	Group not and immediate quadlet
G.NAND.I.64	Group not and immediate octlet
G.NAND.I.128	Group not and immediate hexlet
G.NOR.I.16	Group not or immediate doublet
G.NOR.I.32	Group not or immediate quadlet
G.NOR.I.64	Group not or immediate octlet
G.NOR.I.128	Group not or immediate hexlet
G.OR.I.16	Group or immediate doublet
G.OR.I.32	Group or immediate quadlet
G.OR.I.64	Group or immediate octlet
G.OR.I.128	Group or immediate hexlet
G.XOR.I.16	Group exclusive-or immediate doublet
G.XOR.I.32	Group exclusive-or immediate quadlet
G.XOR.I.64	Group exclusive-or immediate octlet
G.XOR.I.128	Group exclusive-or immediate hexlet

Equivalencies

G.ANDN.I.16	Group and not immediate doublet
G.ANDN.I.32	Group and not immediate quadlet
G.ANDN.I.64	Group and not immediate octlet
G.ANDN.I.128	Group and not immediate hexlet
G.COPY	Group copy
G.NOT	Group not
G.ORN.I.16	Group or not immediate doublet
G.ORN.I.32	Group or not immediate quadlet
G.ORN.I.64	Group or not immediate octlet
G.ORN.I.128	Group or not immediate hexlet
G.XNOR.I.16	Group exclusive-nor immediate doublet
G.XNOR.I.32	Group exclusive-nor immediate quadlet
G.XNOR.I.64	Group exclusive-nor immediate octlet
G.XNOR.I.128	Group exclusive-nor immediate hexlet

G.ANDN.I.size rd=rc.imm	→	G.AND.I.size rd=rc,-imm
G.COPY rd=rc	←	G.OR.I.128 rd=rc,0
G.NOT rd=rc	←	G.NOR.I.128 rd=rc,0
G.ORN.I.size rd=rc.imm	→	G.OR.I.size rd=rc,-imm
G.XNOR.I.size rd=rc.imm	→	G.XOR.I.size rd=rc,-imm

Redundancies

G.ADD.I.size rd=rc,0	⇔	*G.COPY rd=rc*
G.ADD.I.size.O rd=rc,0	⇔	*G.COPY rd=rc*
G.ADD.I.U.size.O rd=rc,0	⇔	*G.COPY rd=rc*
G.AND.I.size rd=rc,0	⇔	*G.ZERO rd*
G.AND.I.size rd=rc,-1	⇔	*G.COPY rd=rc*
G.NAND.I.size rd=rc,0	⇔	*G.SET rd*
G.NAND.I.size rd=rc,-1	⇔	*G.NOT rd=rc*
G.OR.I.size rd=rc,-1	⇔	*G.SET rd*
G.NOR.I.size rd=rc,-1	⇔	*G.ZERO rd*
G.XOR.I.size rd=rc,0	⇔	*G.COPY rd=rc*
G.XOR.I.size rd=rc,-1	⇔	*G.NOT rd=rc*

Format

op.size rd=rc,imm

rd=opsize(rc,imm)

31 … 24	23 … 18	17 … 12	11 10	9 … 0
op	rd	rc	sz	imm
8	6	6	2	10

sz ← log(size)-4

Description

The contents of register rc is fetched, and a 128-bit immediate value is produced from the operation code, the size field and the 10-bit imm field. The specified operation is performed on these operands. The result is placed into register ra.

Definition

```
def GroupImmediate(op,size,rd,rc,imm) as
    c ← RegRead(rc, 128)
    s ← imm_9
    case size of
        16:
            i16 ← s^7 || imm
            b ← i16 || i16 || i16 || i16 || i16 || i16 || i16 || i16
        32:
            b ← s^22 || imm || s^22 || imm || s^22 || imm || s^22 || imm
        64:
            b ← s^54 || imm || s^54 || imm
        128:
            b ← s^118 || imm
    endcase
    case op of
        G.AND.I:
            a ← c and b
        G.OR.I:
            a ← c or b
        G.NAND.I:
            a ← c nand b
        G.NOR.I:
            a ← c nor b
        G.XOR.I:
            a ← c xor b
        G.ADD.I:
            for i ← 0 to 128-size by size
                a_{i+size-1..i} ← c_{i+size-1..i} + b_{i+size-1..i}
            endfor
        G.ADD.I.O:
            for i ← 0 to 128-size by size
                t ← (c_{i+size-1} || c_{i+size-1..i}) + (b_{i+size-1} || b_{i+size-1..i})
```

```
                if t_size ≠ t_size-1 then
                    raise FixedPointArithmetic
                endif
                a_i+size-1..i ← t_size-1..0
            endfor
        G.ADD.I.U.O:
            for i ← 0 to 128-size by size
                t ← (0^1 || c_i+size-1..i) + (0^1 || b_i+size-1..i)
                if t_size ≠ 0 then
                    raise FixedPointArithmetic
                endif
                a_i+size-1..i ← t_size-1..0
            endfor
    endcase
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Fixed-point arithmetic

Group Immediate Reversed

These operations take operands from a register and an immediate value, perform operations on partitions of bits in the operands, and place the concatenated results in a second register.

Operation codes

G.SET.AND.E.I.16	Group set and equal zero immediate doublets
G.SET.AND.E.I.32	Group set and equal zero immediate quadlets
G.SET.AND.E.I.64	Group set and equal zero immediate octlets
G.SET.AND.E.I.128	Group set and equal zero immediate hexlet
G.SET.AND.NE.I.16	Group set and not equal zero immediate doublets
G.SET.AND.NE.I.32	Group set and not equal zero immediate quadlets
G.SET.AND.NE.I.64	Group set and not equal zero immediate octlets
G.SET.AND.NE.I.128	Group set and not equal zero immediate hexlet
G.SET.E.I.16	Group set equal immediate doublets
G.SET.E.I.32	Group set equal immediate quadlets
G.SET.E.I.64	Group set equal immediate octlets
G.SET.E.I.128	Group set equal immediate hexlet
G.SET.GE.I.16	Group set greater equal immediate signed doublets
G.SET.GE.I.32	Group set greater equal immediate signed quadlets
G.SET.GE.I.64	Group set greater equal immediate signed octlets
G.SET.GE.I.128	Group set greater equal immediate signed hexlet
G.SET.GE.I.U.16	Group set greater equal immediate unsigned doublets
G.SET.GE.I.U.32	Group set greater equal immediate unsigned quadlets
G.SET.GE.I.U.64	Group set greater equal immediate unsigned octlets
G.SET.GE.I.U.128	Group set greater equal immediate unsigned hexlet
G.SET.L.I.16	Group set signed less immediate doublets
G.SET.L.I.32	Group set signed less immediate quadlets
G.SET.L.I.64	Group set signed less immediate octlets
G.SET.L.I.128	Group set signed less immediate hexlet
G.SET.L.I.U.16	Group set less immediate signed doublets
G.SET.L.I.U.32	Group set less immediate signed quadlets
G.SET.L.I.U.64	Group set less immediate signed octlets
G.SET.L.I.U.128	Group set less immediate signed hexlet
G.SET.NE.I.16	Group set not equal immediate doublets
G.SET.NE.I.32	Group set not equal immediate quadlets
G.SET.NE.I.64	Group set not equal immediate octlets
G.SET.NE.I.128	Group set not equal immediate hexlet
G.SUB.I.16	Group subtract immediate doublet
G.SUB.I.16.O	Group subtract immediate signed doublet check overflow
G.SUB.I.32	Group subtract immediate quadlet
G.SUB.I.32.O	Group subtract immediate signed quadlet check overflow
G.SUB.I.64	Group subtract immediate octlet
G.SUB.I.64.O	Group subtract immediate signed octlet check overflow
G.SUB.I.128	Group subtract immediate hexlet

G.SUB.I.128.O	Group subtract immediate signed hexlet check overflow
G.SUB.I.U.16.O	Group subtract immediate unsigned doublet check overflow
G.SUB.I.U.32.O	Group subtract immediate unsigned quadlet check overflow
G.SUB.I.U.64.O	Group subtract immediate unsigned octlet check overflow
G.SUB.I.U.128.O	Group subtract immediate unsigned hexlet check overflow

Equivalencies

G.NEG.16	Group negate doublet
G.NEG.16.O	Group negate signed doublet check overflow
G.NEG.32	Group negate quadlet
G.NEG.32.O	Group negate signed quadlet check overflow
G.NEG.64	Group negate octlet
G.NEG.64.O	Group negate signed octlet check overflow
G.NEG.128	Group negate hexlet
G.NEG.128.O	Group negate signed hexlet check overflow
G.SET.LE.I.16	Group set less equal immediate signed doublets
G.SET.LE.I.32	Group set less equal immediate signed quadlets
G.SET.LE.I.64	Group set less equal immediate signed octlets
G.SET.LE.I.128	Group set less equal immediate signed hexlet
G.SET.LE.I.U.16	Group set less equal immediate unsigned doublets
G.SET.LE.I.U.32	Group set less equal immediate unsigned quadlets
G.SET.LE.I.U.64	Group set less equal immediate unsigned octlets
G.SET.LE.I.U.128	Group set less equal immediate unsigned hexlet
G.SET.G.I.16	Group set immediate signed greater doublets
G.SET.G.I.32	Group set immediate signed greater quadlets
G.SET.G.I.64	Group set immediate signed greater octlets
G.SET.G.I.128	Group set immediate signed greater hexlet
G.SET.G.I.U.16	Group set greater immediate unsigned doublets
G.SET.G.I.U.32	Group set greater immediate unsigned quadlets
G.SET.G.I.U.64	Group set greater immediate unsigned octlets
G.SET.G.I.U.128	Group set greater immediate unsigned hexlet

G.NEG.size rd=rc	→	A.SUB.I.size rd=0,rc
G.NEG.size.O rd=rc	→	A.SUB.I.size.O rd=0,rc
G.SET.G.I.size rd=imm,rc	→	G.SET.GE.I.size rd=imm+1,rc
G.SET.G.I.U.size rd=imm,rc	→	G.SET.GE.I.U.size rd=imm+1,rc
G.SET.LE.I.size rd=imm,rc	→	G.SET.L.I.size rd=imm-1,rc
G.SET.LE.I.U.size rd=imm,rc	→	G.SET.L.I.U.size rd=imm-1,rc

Redundancies

G.SET.AND.E.I.size rd=rc,0	⇔	*G.SET.size rd*
G.SET.AND.NE.I.size rd=rc,0	⇔	*G.ZERO rd*
G.SET.AND.E.I.size rd=rc,-1	⇔	*G.SET.E.Z.size rd=rc*
G.SET.AND.NE.I.size rd=rc,-1	⇔	*G.SET.NE.Z.size rd=rc*
G.SET.E.I.size rd=rc,0	⇔	*G.SET.E.Z.size rd=rc*
G.SET.GE.I.size rd=rc,0	⇔	*G.SET.GE.Z.size rd=rc*
G.SET.L.I.size rd=rc,0	⇔	*G.SET.L.Z.size rd=rc*
G.SET.NE.I.size rd=rc,0	⇔	*G.SET.NE.Z.size rd=rc*
G.SET.GE.I.U.size rd=rc,0	⇔	*G.SET.GE.U.Z.size rd=rc*
G.SET.L.I.U.size rd=rc,0	⇔	*G.SET.L.U.Z.size rd=rc*

Selection

class	operation	cond	form	operand	size	check
arithmetic	SUB		I		16 32 64 128	
				NONE U	16 32 64 128	O
boolean	SET.AND SET	E NE	I		16 32 64 128	
	SET	L GE *G LE*	I	NONE U	16 32 64 128	

Format

op.size rd=imm,rc

rd=opsize(imm,rc)

31 24	23 18	17 12	11 10	9 0
op	rd	rc	sz	imm
8	6	6	2	10

sz ← log(size)-4

Description

The contents of register rc is fetched, and a 128-bit immediate value is produced from the operation code, the size field and the 10-bit imm field. The specified operation is performed on these operands. The result is placed into register rd.

Definition

```
def GroupImmediateReversed(op,size,ra,imm) as
    c ← RegRead(rc, 128)
    s ← imm9
    case size of
        16:
```

```
            i16 ← s^7 || imm
            b ← i16 || i16 || i16 || i16 || i16 || i16 || i16 || i16
        32:
            b ← s^22 || imm || s^22 || imm || s^22 || imm || s^22 || imm
        64:
            b ← s^54 || imm || s^54 || imm
        128:
            b ← s^118 || imm
endcase
case op of
    G.SUB.I:
        for i ← 0 to 128-size by size
            a_{i+size-1..i} ← b_{i+size-1..i} - c_{i+size-1..i}
        endfor
    G.SUB.I.O:
        for i ← 0 to 128-size by size
            t ← (b_{i+size-1} || b_{i+size-1..i}) - (c_{i+size-1} || c_{i+size-1..i})
            if (t_size ≠ t_{size-1}) then
                raise FixedPointArithmetic
            endif
            a_{i+size-1..i} ← t_{size-1..0}
        endfor
    G.SUB.I.U.O:
        for i ← 0 to 128-size by size
            t ← (0^1 || b_{i+size-1..i}) - (0^1 || c_{i+size-1..i})
            if (t_size ≠ 0 then
                raise FixedPointArithmetic
            endif
            a_{i+size-1..i} ← t_{size-1..0}
        endfor
    G.SET.E.I:
        for i ← 0 to 128-size by size
            a_{i+size-1..i} ← (b_{i+size-1..i} = c_{i+size-1..i})^size
        endfor
    G.SET.NE.I:
        for i ← 0 to 128-size by size
            a_{i+size-1..i} ← (b_{i+size-1..i} ≠ c_{i+size-1..i})^size
        endfor
    G.SET.AND.E.I:
        for i ← 0 to 128-size by size
            a_{i+size-1..i} ← ((b_{i+size-1..i} and c_{i+size-1..i}) = 0)^size
        endfor
    G.SET.AND.NE.I:
        for i ← 0 to 128-size by size
            a_{i+size-1..i} ← ((b_{i+size-1..i} and c_{i+size-1..i}) ≠ 0)^size
        endfor
    G.SET.L.I:
        for i ← 0 to 128-size by size
            a_{i+size-1..i} ← (b_{i+size-1..i} < c_{i+size-1..i})^size
        endfor
    G.SET.GE.I:
        for i ← 0 to 128-size by size
```

```
                $a_{i+size-1..i}$ ← ($b_{i+size-1..i}$ ≥ $c_{i+size-1..i}$)$^{size}$
            endfor
        G.SET.L.I.U:
            for i ← 0 to 128-size by size
                $a_{i+size-1..i}$ ← ((0 || $b_{i+size-1..i}$) < (0 || $c_{i+size-1..i}$))$^{size}$
            endfor
        G.SET.GE.I.U:
            for i ← 0 to 128-size by size
                $a_{i+size-1..i}$ ← ((0 || $b_{i+size-1..i}$) ≥ (0 || $c_{i+size-1..i}$))$^{size}$
            endfor
    endcase
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Fixed-point arithmetic

Group Inplace

These operations take operands from three registers, perform operations on partitions of bits in the operands, and place the concatenated results in the third register.

Operation codes

G.AAA.8	Group add add add bytes
G.AAA.16	Group add add add doublets
G.AAA.32	Group add add add quadlets
G.AAA.64	Group add add add octlets
G.AAA.128	Group add add add hexlet
G.ASA.8	Group add subtract add bytes
G.ASA.16	Group add subtract add doublets
G.ASA.32	Group add subtract add quadlets
G.ASA.64	Group add subtract add octlets
G.ASA.128	Group add subtract add hexlet

Equivalencies

G.AAS.8	Group add add subtract bytes
G.AAS.16	Group add add subtract doublets
G.AAS.32	Group add add subtract quadlets
G.AAS.64	Group add add subtract octlets
G.AAS.128	Group add add subtract hexlet

G.AAS.size rd@rc,rb	→	G.ASA.size rd@rb,rc

Redundancies

G.AAA.size rd@rc,rc	⇔	G.SHL1ADD.size rd=rd,rc,1
G.ASA.size rd@rc,rc	⇔	*G.NOP*

Format

G.op.size rd@rc,rb

rd=gopsize(rd,rc,rb)

31..24	23..18	17..12	11..6	5..0
G.size	rd	rc	rb	op
8	6	6	6	6

Description

The contents of registers rd, rc and rb are fetched. The specified operation is performed on these operands. The result is placed into register rd.

Register rd is both a source and destination of this instruction.

Definition

```
def GroupInplace(op,size,rd,rc,rb) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    for i ← 0 to 128-size by size
        case op of
            G.AAA:
                a_{i+size-1..i} ← + d_{i+size-1..i} + c_{i+size-1..i} + b_{i+size-1..i}
            G.ASA:
                a_{i+size-1..i} ← + d_{i+size-1..i} - c_{i+size-1..i} + b_{i+size-1..i}
        endcase
    endfor
    RegWrite(rd, 128, a)
enddef
```

Exceptions

none

Group Reversed

These operations take two values from registers, perform operations on partitions of bits in the operands, and place the concatenated results in a register.

Operation codes

G.SET.AND.E.8	Group set and equal zero bytes
G.SET.AND.E.16	Group set and equal zero doublets
G.SET.AND.E.32	Group set and equal zero quadlets
G.SET.AND.E.64	Group set and equal zero octlets
G.SET.AND.E.128	Group set and equal zero hexlet
G.SET.AND.NE.8	Group set and not equal zero bytes
G.SET.AND.NE.16	Group set and not equal zero doublets
G.SET.AND.NE.32	Group set and not equal zero quadlets
G.SET.AND.NE.64	Group set and not equal zero octlets
G.SET.AND.NE.128	Group set and not equal zero hexlet
G.SET.E.8	Group set equal bytes
G.SET.E.16	Group set equal doublets
G.SET.E.32	Group set equal quadlets
G.SET.E.64	Group set equal octlets
G.SET.E.128	Group set equal hexlet
G.SET.GE.8	Group set greater equal signed bytes
G.SET.GE.16	Group set greater equal signed doublets
G.SET.GE.32	Group set greater equal signed quadlets
G.SET.GE.64	Group set greater equal signed octlets
G.SET.GE.128	Group set greater equal signed hexlet
G.SET.GE.U.8	Group set greater equal unsigned bytes
G.SET.GE.U.16	Group set greater equal unsigned doublets
G.SET.GE.U.32	Group set greater equal unsigned quadlets
G.SET.GE.U.64	Group set greater equal unsigned octlets
G.SET.GE.U.128	Group set greater equal unsigned hexlet
G.SET.L.8	Group set signed less bytes
G.SET.L.16	Group set signed less doublets
G.SET.L.32	Group set signed less quadlets
G.SET.L.64	Group set signed less octlets
G.SET.L.128	Group set signed less hexlet
G.SET.L.U.8	Group set less unsigned bytes
G.SET.L.U.16	Group set less unsigned doublets
G.SET.L.U.32	Group set less unsigned quadlets
G.SET.L.U.64	Group set less unsigned octlets
G.SET.L.U.128	Group set less unsigned hexlet
G.SET.NE.8	Group set not equal bytes
G.SET.NE.16	Group set not equal doublets
G.SET.NE.32	Group set not equal quadlets
G.SET.NE.64	Group set not equal octlets

G.SET.NE.128	Group set not equal hexlet
G.SUB.8	Group subtract bytes
G.SUB.8.O	Group subtract signed bytes check overflow
G.SUB.16	Group subtract doublets
G.SUB.16.O	Group subtract signed doublets check overflow
G.SUB.32	Group subtract quadlets
G.SUB.32.O	Group subtract signed quadlets check overflow
G.SUB.64	Group subtract octlets
G.SUB.64.O	Group subtract signed octlets check overflow
G.SUB.128	Group subtract hexlet
G.SUB.128.O	Group subtract signed hexlet check overflow
G.SUB.L.8	Group subtract limit signed bytes
G.SUB.L.16	Group subtract limit signed doublets
G.SUB.L.32	Group subtract limit signed quadlets
G.SUB.L.64	Group subtract limit signed octlets
G.SUB.L.128	Group subtract limit signed hexlet
G.SUB.L.U.8	Group subtract limit unsigned bytes
G.SUB.L.U.16	Group subtract limit unsigned doublets
G.SUB.L.U.32	Group subtract limit unsigned quadlets
G.SUB.L.U.64	Group subtract limit unsigned octlets
G.SUB.L.U.128	Group subtract limit unsigned hexlet
G.SUB.U.8.O	Group subtract unsigned bytes check overflow
G.SUB.U.16.O	Group subtract unsigned doublets check overflow
G.SUB.U.32.O	Group subtract unsigned quadlets check overflow
G.SUB.U.64.O	Group subtract unsigned octlets check overflow
G.SUB.U.128.O	Group subtract unsigned hexlet check overflow

Equivalencies

G.SET.E.Z.8	Group set equal zero bytes
G.SET.E.Z.16	Group set equal zero doublets
G.SET.E.Z.32	Group set equal zero quadlets
G.SET.E.Z.64	Group set equal zero octlets
G.SET.E.Z.128	Group set equal zero hexlet
G.SET.G.Z.8	Group set greater zero signed bytes
G.SET.G.Z.16	Group set greater zero signed doublets
G.SET.G.Z.32	Group set greater zero signed quadlets
G.SET.G.Z.64	Group set greater zero signed octlets
G.SET.G.Z.128	Group set greater zero signed hexlet
G.SET.GE.Z.8	Group set greater equal zero signed bytes
G.SET.GE.Z.16	Group set greater equal zero signed doublets
G.SET.GE.Z.32	Group set greater equal zero signed quadlets
G.SET.GE.Z.64	Group set greater equal zero signed octlets
G.SET.GE.Z.128	Group set greater equal zero signed hexlet
G.SET.L.Z.8	Group set less zero signed bytes
G.SET.L.Z.16	Group set less zero signed doublets
G.SET.L.Z.32	Group set less zero signed quadlets
G.SET.L.Z.64	Group set less zero signed octlets
G.SET.L.Z.128	Group set less zero signed hexlet
G.SET.LE.Z.8	Group set less equal zero signed bytes
G.SET.LE.Z.16	Group set less equal zero signed doublets
G.SET.LE.Z.32	Group set less equal zero signed quadlets
G.SET.LE.Z.64	Group set less equal zero signed octlets
G.SET.LE.Z.128	Group set less equal zero signed hexlet
G.SET.NE.Z.8	Group set not equal zero bytes
G.SET.NE.Z.16	Group set not equal zero doublets
G.SET.NE.Z.32	Group set not equal zero quadlets
G.SET.NE.Z.64	Group set not equal zero octlets
G.SET.NE.Z.128	Group set not equal zero hexlet
G.SET.LE.8	Group set less equal signed bytes
G.SET.LE.16	Group set less equal signed doublets
G.SET.LE.32	Group set less equal signed quadlets
G.SET.LE.64	Group set less equal signed octlets
G.SET.LE.128	Group set less equal signed hexlet
G.SET.LE.U.8	Group set less equal unsigned bytes
G.SET.LE.U.16	Group set less equal unsigned doublets
G.SET.LE.U.32	Group set less equal unsigned quadlets
G.SET.LE.U.64	Group set less equal unsigned octlets
G.SET.LE.U.128	Group set less equal unsigned hexlet
G.SET.G.8	Group set signed greater bytes
G.SET.G.16	Group set signed greater doublets
G.SET.G.32	Group set signed greater quadlets
G.SET.G.64	Group set signed greater octlets

G.SET.G.128	Group set signed greater hexlet
G.SET.G.U.8	Group set greater unsigned bytes
G.SET.G.U.16	Group set greater unsigned doublets
G.SET.G.U.32	Group set greater unsigned quadlets
G.SET.G.U.64	Group set greater unsigned octlets
G.SET.G.U.128	Group set greater unsigned hexlet

G.SET.E.Z.size rd=rc	←	G.SET.AND.E.size rd=rc,rc
G.SET.G.Z.size rd=rc	⇐	G.SET.L.U.size rd=rc,rc
G.SET.GE.Z.size rd=rc	⇐	G.SET.GE.size rd=rc,rc
G.SET.L.Z.size rd=rc	⇐	G.SET.L.size rd=rc,rc
G.SET.LE.Z.size rd=rc	⇐	G.SET.GE.U.size rd=rc,rc
G.SET.NE.Z.size rd=rc	←	G.SET.AND.NE.size rd=rc,rc
G.SET.G.size rd=rb,rc	→	G.SET.L.size rd=rc,rb
G.SET.G.U.size rd=rb,rc	→	G.SET.L.U.size rd=rc,rb
G.SET.LE.size rd=rb,rc	→	G.SET.GE.size rd=rc,rb
G.SET.LE.U.size rd=rb,rc	→	G.SET.GE.U.size rd=rc,rb

Redundancies

G.SET.E.size rd=rc,rc	⇔	G.SET rd
G.SET.NE.size rd=rc,rc	⇔	G.ZERO rd
G.SUB.size rd=rc,rc	⇔	G.ZERO rd
G.SUB.L.size rd=rc,rc	⇔	G.ZERO rd
G.SUB.L.U.size rd=rc,rc	⇔	G.ZERO rd
G.SUB.size.O rd=rc,rc	⇔	G.ZERO rd
G.SUB.U.size.O rd=rc,rc	⇔	G.ZERO rd

Selection

class	operation	cond	operand	size	check
arithmetic	SUB			8 16 32 64 128	
			NONE U	8 16 32 64 128	O
	SUB.L		NONE U	8 16 32 64 128	
boolean	SET.AND SET	E NE		8 16 32 64 128	
	SET	L GE G LE	NONE U	8 16 32 64 128	
	SET	G GE L LE	Z	8 16 32 64 128	

Format

G.op.size rd=rb,rc

rd=gopsize(rb,rc)

31 … 24	23 … 18	17 … 12	11 … 6	5 … 0
G.size	rd	rc	rb	op
8	6	6	6	6

Description

Two values are taken from the contents of registers rc and rb. The specified operation is performed, and the result is placed in register rd.

Definition

```
def GroupReversed(op,size,rd,rc,rb)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    case op of
        G.SUB:
            for i ← 0 to 128-size by size
                a[i+size-1..i] ← b[i+size-1..i] - c[i+size-1..i]
            endfor
        G.SUB.L:
            for i ← 0 to 128-size by size
                t ← (b[i+size-1] || b[i+size-1..i]) - (c[i+size-1] || c[i+size-1..i])
                a[i+size-1..i] ← (t[size] ≠ t[size-1]) ? (t[size] || ~t[size]^(size-1)) : t[size-1..0]
            endfor
        G.SUB.L.U:
            for i ← 0 to 128-size by size
                t ← (0^1 || b[i+size-1..i]) - (0^1 || c[i+size-1..i])
                a[i+size-1..i] ← (t[size] ≠ 0) ? 0^size : t[size-1..0]
            endfor
        G.SUB.O:
            for i ← 0 to 128-size by size
                t ← (b[i+size-1] || b[i+size-1..i]) - (c[i+size-1] || c[i+size-1..i])
                if (t[size] ≠ t[size-1]) then
                    raise FixedPointArithmetic
                endif
                a[i+size-1..i] ← t[size-1..0]
            endfor
        G.SUB.U.O:
            for i ← 0 to 128-size by size
                t ← (0^1 || b[i+size-1..i]) - (0^1 || c[i+size-1..i])
                if (t[size] ≠ 0) then
                    raise FixedPointArithmetic
                endif
                a[i+size-1..i] ← t[size-1..0]
            endfor
        G.SET.E:
```

```
            for i ← 0 to 128-size by size
                a[i+size-1..i] ← (b[i+size-1..i] = c[i+size-1..i])^size
            endfor
        G.SET.NE:
            for i ← 0 to 128-size by size
                a[i+size-1..i] ← (b[i+size-1..i] ≠ c[i+size-1..i])^size
            endfor
        G.SET.AND.E:
            for i ← 0 to 128-size by size
                a[i+size-1..i] ← ((b[i+size-1..i] and c[i+size-1..i]) = 0)^size
            endfor
        G.SET.AND.NE:
            for i ← 0 to 128-size by size
                a[i+size-1..i] ← ((b[i+size-1..i] and c[i+size-1..i]) ≠ 0)^size
            endfor
        G.SET.L:
            for i ← 0 to 128-size by size
                a[i+size-1..i] ← ((rc = rb) ? (b[i+size-1..i] < 0) : (b[i+size-1..i] < c[i+size-1..i]))^size
            endfor
        G.SET.GE:
            for i ← 0 to 128-size by size
                a[i+size-1..i] ← ((rc = rb) ? (b[i+size-1..i] ≥ 0) : (b[i+size-1..i] ≥ c[i+size-1..i]))^size
            endfor
        G.SET.L.U:
            for i ← 0 to 128-size by size
                a[i+size-1..i] ← ((rc = rb) ? (b[i+size-1..i] > 0) :
                    ((0 || b[i+size-1..i]) < (0 || c[i+size-1..i])))^size
            endfor
        G.SET.GE.U:
            for i ← 0 to 128-size by size
                a[i+size-1..i] ← ((rc = rb) ? (b[i+size-1..i] ≤ 0) :
                    ((0 || b[i+size-1..i]) ≥ (0 || c[i+size-1..i])))^size
            endfor
    endcase
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Fixed-point arithmetic

Group Reversed Floating-point

These operations take two values from registers, perform a group of floating-point arithmetic operations on partitions of bits in the operands, and place the concatenated results in a register.

Operation codes

G.SET.E.F.16	Group set equal floating-point half
G.SET.E.F.16.X	Group set equal floating-point half exact
G.SET.E.F.32	Group set equal floating-point single
G.SET.E.F.32.X	Group set equal floating-point single exact
G.SET.E.F.64	Group set equal floating-point double
G.SET.E.F.64.X	Group set equal floating-point double exact
G.SET.E.F.128	Group set equal floating-point quad
G.SET.E.F.128.X	Group set equal floating-point quad exact
G.SET.GE.F.16.X	Group set greater equal floating-point half exact
G.SET.GE.F.32.X	Group set greater equal floating-point single exact
G.SET.GE.F.64.X	Group set greater equal floating-point double exact
G.SET.GE.F.128.X	Group set greater equal floating-point quad exact
G.SET.LG.F.16	Group set less greater floating-point half
G.SET.LG.F.16.X	Group set less greater floating-point half exact
G.SET.LG.F.32	Group set less greater floating-point single
G.SET.LG.F.32.X	Group set less greater floating-point single exact
G.SET.LG.F.64	Group set less greater floating-point double
G.SET.LG.F.64.X	Group set less greater floating-point double exact
G.SET.LG.F.128	Group set less greater floating-point quad
G.SET.LG.F.128.X	Group set less greater floating-point quad exact
G.SET.L.F.16	Group set less floating-point half
G.SET.L.F.16.X	Group set less floating-point half exact
G.SET.L.F.32	Group set less floating-point single
G.SET.L.F.32.X	Group set less floating-point single exact
G.SET.L.F.64	Group set less floating-point double
G.SET.L.F.64.X	Group set less floating-point double exact
G.SET.L.F.128	Group set less floating-point quad
G.SET.L.F.128.X	Group set less floating-point quad exact
G.SET.GE.F.16	Group set greater equal floating-point half
G.SET.GE.F.32	Group set greater equal floating-point single
G.SET.GE.F.64	Group set greater equal floating-point double
G.SET.GE.F.128	Group set greater equal floating-point quad

Equivalencies

G.SET.LE.F.16X	Group set less equal floating-point half exact
G.SET.LE.F.32X	Group set less equal floating-point single exact
G.SET.LE.F.64X	Group set less equal floating-point double exact
G.SET.LE.F.128X	Group set less equal floating-point quad exact
G.SET.G.F.16	Group set greater floating-point half
G.SET.G.F.16X	Group set greater floating-point half exact
G.SET.G.F.32	Group set greater floating-point single
G.SET.G.F.32X	Group set greater floating-point single exact
G.SET.G.F.64	Group set greater floating-point double
G.SET.G.F.64X	Group set greater floating-point double exact
G.SET.G.F.128	Group set greater floating-point quad
G.SET.G.F.128X	Group set greater floating-point quad exact
G.SET.LE.F.16	Group set less equal floating-point half
G.SET.LE.F.32	Group set less equal floating-point single
G.SET.LE.F.64	Group set less equal floating-point double
G.SET.LE.F.128	Group set less equal floating-point quad

G.SET.G.F.prec rd=rb,rc	→	G.SET.L.F.prec rd=rc,rb
G.SET.G.F.precX rd=rb,rc	→	G.SET.L.F.precX rd=rc,rb
G.SET.LE.F.prec rd=rb,rc	→	G.SET.GE.F.prec rd=rc,rb
G.SET.LE.F.precX rd=rb,rc	→	G.SET.GE.F.precX rd=rc,rb

Selection

class	op	prec	round/trap
set	SET. E LG L GE *G LE*	16 32 64 128	NONE X

Format

G.op.prec.round rd=rb,rc

rc=gopprecround(rb,ra)

31 … 24	23 … 18	17 … 12	11 … 6	5 … 0
G.prec	rd	rc	rb	op.round
8	6	6	6	6

Description

The contents of registers ra and rb are combined using the specified floating-point operation. The result is placed in register rc. The operation is rounded using the specified rounding option or using round-to-nearest if not specified. If a rounding option is specified, the operation raises a floating-point exception if a floating-point invalid operation, divide by

zero, overflow, or underflow occurs, or when specified, if the result is inexact. If a rounding option is not specified, floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754.

Definition

```
def GroupFloatingPointReversed(op,prec,round,rd,rc,rb) as
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    for i ← 0 to 128-prec by prec
        ci ← F(prec,c[i+prec-1..i])
        bi ← F(prec,b[i+prec-1..i])
        if round≠NONE then
            if (di.t = SNAN) or (ci.t = SNAN) then
                raise FloatingPointArithmetic
            endif
            case op of
                G.SET.L.F, G.SET.GE.F:
                    if (di.t = QNAN) or (ci.t = QNAN) then
                        raise FloatingPointArithmetic
                    endif
                others: //nothing
            endcase
        endif
        case op of
            G.SET.L.F:
                ai ← bi?≥ci
            G.SET.GE.F:
                ai ← bi!?<ci
            G.SET.E.F:
                ai ← bi=ci
            G.SET.LG.F:
                ai ← bi≠ci
        endcase
        a[i+prec-1..i] ← ai
    endfor
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Floating-point arithmetic

Group Shift Left Immediate Add

These operations take operands from two registers, perform operations on partitions of bits in the operands, and place the concatenated results in a third register.

Operation codes

G.SHL.I.ADD.8	Group shift left immediate add bytes
G.SHL.I.ADD.16	Group shift left immediate add doublets
G.SHL.I.ADD.32	Group shift left immediate add quadlets
G.SHL.I.ADD.64	Group shift left immediate add octlets
G.SHL.I.ADD.128	Group shift left immediate add hexlet

Redundancies

G.SHL.I.ADD.size rd=rd,rc,1	⇔	G.AAA.size rd@rc,rc

Format

G.op.size rd=rc,rb,i

rd=gopsize(rc,rb,i)

31 24	23 18	17 12	11 6	5 2	1 0
G.size	rd	rc	rb	GSHLIADD	sh
8	6	6	6	6	2

assert $1 \le i \le 4$
sh ← i-1

Description

The contents of registers rc and rb are partitioned into groups of operands of the size specified. Partitions of the contents of register rb are shifted left by the amount specified in the immediate field and added to partitions of the contents of register rc, yielding a group of results, each of which is the size specified. Overflows are ignored, and yield modular arithmetic results. The group of results is catenated and placed in register rd.

Definition

```
def GroupShiftLeftImmediateAdd(sh,size,ra,rb,rc)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    for i ← 0 to 128-size by size
        a[i+size-1..i] ← c[i+size-1..i] + (b[i+size-1-sh..i] || 0^(1+sh))
    endfor
    RegWrite(rd, 128, a)
enddef
```

Exceptions

none

Group Shift Left Immediate Subtract

These operations take operands from two registers, perform operations on partitions of bits in the operands, and place the concatenated results in a third register.

Operation codes

G.SHL.I.SUB.8	Group shift left immediate subtract bytes
G.SHL.I.SUB.16	Group shift left immediate subtract doublets
G.SHL.I.SUB.32	Group shift left immediate subtract quadlets
G.SHL.I.SUB.64	Group shift left immediate subtract octlets
G.SHL.I.SUB.128	Group shift left immediate subtract hexlet

Redundancies

G.SHL.I.SUB.size rd=rc,1,rc	⇔	G.COPY rd=rc

Format

G.op.size rd=rb,i,rc

rd=gopsize(rb,i,rc)

31..24	23..18	17..12	11..6	5..2	1..0
G.size	rd	rc	rb	GSHLISUB	sh
8	6	6	6	6	2

assert 1≤i≤4
sh ← i-1

Description

The contents of registers rc and rb are partitioned into groups of operands of the size specified. Partitions of the contents of register rc are subtracted from partitions of the contents of register rb shifted left by the amount specified in the immediate field, yielding a group of results, each of which is the size specified. Overflows are ignored, and yield modular arithmetic results. The group of results is catenated and placed in register rd.

Definition

```
def GroupShiftLeftImmediateSubtract(sh,size,ra,rb,rc)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    for i ← 0 to 128-size by size
        a[i+size-1..i] ← (b[i+size-1-sh..i] || 0^(1+sh)) - c[i+size-1..i]
    endfor
    RegWrite(rd, 128, a)
enddef
```

Exceptions

none

Group Subtract Halve

These operations take operands from two registers, perform operations on partitions of bits in the operands, and place the concatenated results in a third register.

Operation codes

G.SUB.H.8.C	Group subtract halve signed bytes ceiling
G.SUB.H.8.F	Group subtract halve signed bytes floor
G.SUB.H.8.N	Group subtract halve signed bytes nearest
G.SUB.H.8.Z	Group subtract halve signed bytes zero
G.SUB.H.16.C	Group subtract halve signed doublets ceiling
G.SUB.H.16.F	Group subtract halve signed doublets floor
G.SUB.H.16.N	Group subtract halve signed doublets nearest
G.SUB.H.16.Z	Group subtract halve signed doublets zero
G.SUB.H.32.C	Group subtract halve signed quadlets ceiling
G.SUB.H.32.F	Group subtract halve signed quadlets floor
G.SUB.H.32.N	Group subtract halve signed quadlets nearest
G.SUB.H.32.Z	Group subtract halve signed quadlets zero
G.SUB.H.64.C	Group subtract halve signed octlets ceiling
G.SUB.H.64.F	Group subtract halve signed octlets floor
G.SUB.H.64.N	Group subtract halve signed octlets nearest
G.SUB.H.64.Z	Group subtract halve signed octlets zero
G.SUB.H.128.C	Group subtract halve signed hexlet ceiling
G.SUB.H.128.F	Group subtract halve signed hexlet floor
G.SUB.H.128.N	Group subtract halve signed hexlet nearest
G.SUB.H.128.Z	Group subtract halve signed hexlet zero
G.SUB.H.U.8.C	Group subtract halve unsigned bytes ceiling
G.SUB.H.U.8.F	Group subtract halve unsigned bytes floor
G.SUB.H.U.8.N	Group subtract halve unsigned bytes nearest
G.SUB.H.U.8.Z	Group subtract halve unsigned bytes zero
G.SUB.H.U.16.C	Group subtract halve unsigned doublets ceiling
G.SUB.H.U.16.F	Group subtract halve unsigned doublets floor
G.SUB.H.U.16.N	Group subtract halve unsigned doublets nearest
G.SUB.H.U.16.Z	Group subtract halve unsigned doublets zero
G.SUB.H.U.32.C	Group subtract halve unsigned quadlets ceiling
G.SUB.H.U.32.F	Group subtract halve unsigned quadlets floor
G.SUB.H.U.32.N	Group subtract halve unsigned quadlets nearest
G.SUB.H.U.32.Z	Group subtract halve unsigned quadlets zero
G.SUB.H.U.64.C	Group subtract halve unsigned octlets ceiling
G.SUB.H.U.64.F	Group subtract halve unsigned octlets floor
G.SUB.H.U.64.N	Group subtract halve unsigned octlets nearest
G.SUB.H.U.64.Z	Group subtract halve unsigned octlets zero
G.SUB.H.U.128.C	Group subtract halve unsigned hexlet ceiling
G.SUB.H.U.128.F	Group subtract halve unsigned hexlet floor
G.SUB.H.U.128.N	Group subtract halve unsigned hexlet nearest

G.SUB.H.U.128.Z	Group subtract halve unsigned hexlet zero

Redundancies

G.SUB.H.size.rnd rd=rc,rc	⇔	G.ZERO rd
G.SUB.H.U.size.rnd rd=rc,rc	⇔	G.ZERO rd

Format

G.op.size.rnd rd=rb,rc

rd=gopsizernd(rb,rc)

31..24	23..18	17..12	11..6	5..2	1..0
G.size	rd	rc	rb	op	rnd
8	6	6	6	4	2

Description

The contents of registers rc and rb are partitioned into groups of operands of the size specified and subtracted, halved, rounded and limited as specified, yielding a group of results, each of which is the size specified. The group of results is catenated and placed in register rd.

The result of this operation is always signed, whether the operands are signed or unsigned.

Definition

```
def GroupSubtractHalve(op,rnd,size,rd,rc,rb)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    case op of
        G.SUB.H.C, G.SUB.H.F, G.SUB.H.N, G.SUB.H.Z:
            as ← cs ← bs ← 1
        G.SUB.H.U.C, G.SUB.H.U.F, G.SUB.H.U.N, G.SUB.H.U.Z
            as ← 1
            cs ← bs ← 0
    endcase
    for i ← 0 to 128-size by size
        p ← ((bs and b_{size-1}) || b_{size-1+i..i}) - ((cs and c_{size-1}) || c_{size-1+i..i})
        case rnd of
            none, N:
                s ← 0^{size} || -p_1
            Z:
                s ← 0^{size} || p_{size}
            F:
                s ← 0^{size+1}
            C:
                s ← 0^{size} || 1^1
        endcase
```

```
            v ← ((as & p_size) || p) + (0 || s)
            if v_size+1 = (as & v_size) then
                a_size-1+i..i ← v_size..1
            else
                a_size-1+i..i ← as ? (v_size+1 || ~v^size-1_size+1) : 1^size
            endif
        endfor
        RegWrite(rd, 128, a)
enddef
```

Exceptions

none

Group Ternary

These operations take three values from registers, perform a group of calculations on partitions of bits of the operands and place the catenated results in a fourth register.

Operation codes

G.MUX	Group multiplex

Redundancies

G.MUX ra=rd,rc,rc	⇔	*G.COPY ra=rc*
G.MUX ra=ra,rc,rb	⇔	G.BOOLEAN ra@rc,rb,0x11001010
G.MUX ra=rd,ra,rb	⇔	G.BOOLEAN ra@rd,rb,0x11100010
G.MUX ra=rd,rc,ra	⇔	G.BOOLEAN ra@rd,rc,0x11011000
G.MUX ra=rd,rd,rb	⇔	G.OR ra=rd,rb
G.MUX ra=rd,rc,rd	⇔	G.AND ra=rd,rc

Format

G.MUX ra=rd,rc,rb

ra=gmux(rd,rc,rb)

31 24	23 18	17 12	11 6	5 0
G.MUX	rd	rc	rb	ra
8	6	6	6	6

Description

The contents of registers rd, rc, and rb are fetched. Each bit of the result is equal to the corresponding bit of rc, if the corresponding bit of rd is set, otherwise it is the corresponding bit of rb. The result is placed into register ra.

Definition

```
def GroupTernary(op,size,rd,rc,rb,ra) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    case op of
        G.MUX:
            a ← (c and d) or (b and not d)
    endcase
    RegWrite(ra, 128, a)
enddef
```

Exceptions

none

Crossbar

These operations take operands from two registers, perform operations on partitions of bits in the operands, and place the concatenated results in a third register.

Operation codes

X.COMPRESS.2	Crossbar compress signed pecks
X.COMPRESS.4	Crossbar compress signed nibbles
X.COMPRESS.8	Crossbar compress signed bytes
X.COMPRESS.16	Crossbar compress signed doublets
X.COMPRESS.32	Crossbar compress signed quadlets
X.COMPRESS.64	Crossbar compress signed octlets
X.COMPRESS.128	Crossbar compress signed hexlet
X.COMPRESS.U.2	Crossbar compress unsigned pecks
X.COMPRESS.U.4	Crossbar compress unsigned nibbles
X.COMPRESS.U.8	Crossbar compress unsigned bytes
X.COMPRESS.U.16	Crossbar compress unsigned doublets
X.COMPRESS.U.32	Crossbar compress unsigned quadlets
X.COMPRESS.U.64	Crossbar compress unsigned octlets
X.COMPRESS.U.128	Crossbar compress unsigned hexlet
X.EXPAND.2	Crossbar expand signed pecks
X.EXPAND.4	Crossbar expand signed nibbles
X.EXPAND.8	Crossbar expand signed bytes
X.EXPAND.16	Crossbar expand signed doublets
X.EXPAND.32	Crossbar expand signed quadlets
X.EXPAND.64	Crossbar expand signed octlets
X.EXPAND.128	Crossbar expand signed hexlet
X.EXPAND.U.2	Crossbar expand unsigned pecks
X.EXPAND.U.4	Crossbar expand unsigned nibbles
X.EXPAND.U.8	Crossbar expand unsigned bytes
X.EXPAND.U.16	Crossbar expand unsigned doublets
X.EXPAND.U.32	Crossbar expand unsigned quadlets
X.EXPAND.U.64	Crossbar expand unsigned octlets
X.EXPAND.U.128	Crossbar expand unsigned hexlet
X.ROTL.2	Crossbar rotate left pecks
X.ROTL.4	Crossbar rotate left nibbles
X.ROTL.8	Crossbar rotate left bytes
X.ROTL.16	Crossbar rotate left doublets
X.ROTL.32	Crossbar rotate left quadlets
X.ROTL.64	Crossbar rotate left octlets
X.ROTL.128	Crossbar rotate left hexlet
X.ROTR.2	Crossbar rotate right pecks
X.ROTR.4	Crossbar rotate right nibbles
X.ROTR.8	Crossbar rotate right bytes
X.ROTR.16	Crossbar rotate right doublets
X.ROTR.32	Crossbar rotate right quadlets

X.ROTR.64	Crossbar rotate right octlets
X.ROTR.128	Crossbar rotate right hexlet
X.SHL.2	Crossbar shift left pecks
X.SHL.2.O	Crossbar shift left signed pecks check overflow
X.SHL.4	Crossbar shift left nibbles
X.SHL.4.O	Crossbar shift left signed nibbles check overflow
X.SHL.8	Crossbar shift left bytes
X.SHL.8.O	Crossbar shift left signed bytes check overflow
X.SHL.16	Crossbar shift left doublets
X.SHL.16.O	Crossbar shift left signed doublets check overflow
X.SHL.32	Crossbar shift left quadlets
X.SHL.32.O	Crossbar shift left signed quadlets check overflow
X.SHL.64	Crossbar shift left octlets
X.SHL.64.O	Crossbar shift left signed octlets check overflow
X.SHL.128	Crossbar shift left hexlet
X.SHL.128.O	Crossbar shift left signed hexlet check overflow
X.SHL.U.2.O	Crossbar shift left unsigned pecks check overflow
X.SHL.U.4.O	Crossbar shift left unsigned nibbles check overflow
X.SHL.U.8.O	Crossbar shift left unsigned bytes check overflow
X.SHL.U.16.O	Crossbar shift left unsigned doublets check overflow
X.SHL.U.32.O	Crossbar shift left unsigned quadlets check overflow
X.SHL.U.64.O	Crossbar shift left unsigned octlets check overflow
X.SHL.U.128.O	Crossbar shift left unsigned hexlet check overflow
X.SHR.2	Crossbar signed shift right pecks
X.SHR.4	Crossbar signed shift right nibbles
X.SHR.8	Crossbar signed shift right bytes
X.SHR.16	Crossbar signed shift right doublets
X.SHR.32	Crossbar signed shift right quadlets
X.SHR.64	Crossbar signed shift right octlets
X.SHR.128	Crossbar signed shift right hexlet
X.SHR.U.2	Crossbar shift right unsigned pecks
X.SHR.U.4	Crossbar shift right unsigned nibbles
X.SHR.U.8	Crossbar shift right unsigned bytes
X.SHR.U.16	Crossbar shift right unsigned doublets
X.SHR.U.32	Crossbar shift right unsigned quadlets
X.SHR.U.64	Crossbar shift right unsigned octlets
X.SHR.U.128	Crossbar shift right unsigned hexlet

Selection

class	op	size
precision	EXPAND EXPAND.U COMPRESS COMPRESS.U	2 4 8 16 32 64 128
shift	ROTR ROTL SHR SHL SHL.O SHL.U.O SHR.U	2 4 8 16 32 64 128

Format

X.op.size rd=rc,rb

rd=xopsize(rc,rb)

31 ... 25	24	23 ... 18	17 ... 12	11 ... 6	5 ... 2	1 ... 0
XSHIFT	s	rd	rc	rb	op	sz
7	1	6	6	6	4	2

$$
\begin{array}{l}
\text{lsize} \leftarrow \log(\text{size}) \\
s \leftarrow \text{lsize}_2 \\
sz \leftarrow \text{lsize}_{1..0}
\end{array}
$$

Description

Two values are taken from the contents of registers rc and rb. The specified operation is performed, and the result is placed in register rd.

Definition

$$
\begin{array}{l}
\textit{def}\ \text{Crossbar(op,size,rd,rc,rb)} \\
\quad c \leftarrow \text{RegRead(rc, 128)} \\
\quad b \leftarrow \text{RegRead(rb, 128)} \\
\quad \text{shift} \leftarrow b\ \textit{and}\ (\text{size}-1) \\
\quad \textit{case}\ \text{op}_{5..2} \mid\mid 0^2\ \textit{of} \\
\quad\quad \text{X.COMPRESS:} \\
\quad\quad\quad \text{hsize} \leftarrow \text{size}/2 \\
\quad\quad\quad \textit{for}\ i \leftarrow 0\ \textit{to}\ 64\text{-hsize}\ \textit{by}\ \text{hsize} \\
\quad\quad\quad\quad \textit{if}\ \text{shift} \le \text{hsize}\ \textit{then} \\
\quad\quad\quad\quad\quad a_{i+\text{hsize}-1..i} \leftarrow c_{i+i+\text{shift}+\text{hsize}-1..i+i+\text{shift}} \\
\quad\quad\quad\quad \textit{else} \\
\quad\quad\quad\quad\quad a_{i+\text{hsize}-1..i} \leftarrow c_{i+i+\text{size}-1}^{\text{shift}-\text{hsize}} \mid\mid c_{i+i+\text{size}-1..i+i+\text{shift}} \\
\quad\quad\quad\quad \textit{endif} \\
\quad\quad\quad \textit{endfor} \\
\quad\quad\quad a_{127..64} \leftarrow 0 \\
\quad\quad \text{X.COMPRESS.U:} \\
\quad\quad\quad \text{hsize} \leftarrow \text{size}/2 \\
\quad\quad\quad \textit{for}\ i \leftarrow 0\ \textit{to}\ 64\text{-hsize}\ \textit{by}\ \text{hsize} \\
\quad\quad\quad\quad \textit{if}\ \text{shift} \le \text{hsize}\ \textit{then} \\
\quad\quad\quad\quad\quad a_{i+\text{hsize}-1..i} \leftarrow c_{i+i+\text{shift}+\text{hsize}-1..i+i+\text{shift}} \\
\quad\quad\quad\quad \textit{else} \\
\quad\quad\quad\quad\quad a_{i+\text{hsize}-1..i} \leftarrow 0^{\text{shift}-\text{hsize}} \mid\mid c_{i+i+\text{size}-1..i+i+\text{shift}} \\
\quad\quad\quad\quad \textit{endif} \\
\quad\quad\quad \textit{endfor} \\
\quad\quad\quad a_{127..64} \leftarrow 0 \\
\quad\quad \text{X.EXPAND:} \\
\quad\quad\quad \text{hsize} \leftarrow \text{size}/2 \\
\quad\quad\quad \textit{for}\ i \leftarrow 0\ \textit{to}\ 64\text{-hsize}\ \textit{by}\ \text{hsize} \\
\quad\quad\quad\quad \textit{if}\ \text{shift} \le \text{hsize}\ \textit{then} \\
\quad\quad\quad\quad\quad a_{i+i+\text{size}-1..i+i} \leftarrow c_{i+\text{hsize}-1}^{\text{hsize}-\text{shift}} \mid\mid c_{i+\text{hsize}-1..i} \mid\mid 0^{\text{shift}}
\end{array}
$$

```
            else
                a_{i+i+size-1..i+i} ← c_{i+size-shift-1..i} || 0^shift
            endif
        endfor
    X.EXPAND.U:
        hsize ← size/2
        for i ← 0 to 64-hsize by hsize
            if shift ≤ hsize then
                a_{i+i+size-1..i+i} ← 0^(hsize-shift) || c_{i+hsize-1..i} || 0^shift
            else
                a_{i+i+size-1..i+i} ← c_{i+size-shift-1..i} || 0^shift
            endif
        endfor
    X.ROTL:
        for i ← 0 to 128-size by size
            a_{i+size-1..i} ← c_{i+size-1-shift..i} || c_{i+size-1..i+size-1-shift}
        endfor
    X.ROTR:
        for i ← 0 to 128-size by size
            a_{i+size-1..i} ← c_{i+shift-1..i} || c_{i+size-1..i+shift}
        endfor
    X.SHL:
        for i ← 0 to 128-size by size
            a_{i+size-1..i} ← c_{i+size-1-shift..i} || 0^shift
        endfor
    X.SHL.O:
        for i ← 0 to 128-size by size
            if c_{i+size-1..i+size-1-shift} ≠ c_{i+size-1-shift}^(shift+1) then
                raise FixedPointArithmetic
            endif
            a_{i+size-1..i} ← c_{i+size-1-shift..i} || 0^shift
        endfor
    X.SHL.U.O:
        for i ← 0 to 128-size by size
            if c_{i+size-1..i+size-shift} ≠ 0^shift then
                raise FixedPointArithmetic
            endif
            a_{i+size-1..i} ← c_{i+size-1-shift..i} || 0^shift
        endfor
    X.SHR:
        for i ← 0 to 128-size by size
            a_{i+size-1..i} ← c_{i+size-1}^shift || c_{i+size-1..i+shift}
        endfor
    X.SHR.U:
        for i ← 0 to 128-size by size
            a_{i+size-1..i} ← 0^shift || c_{i+size-1..i+shift}
        endfor
endcase
RegWrite(rd, 128, a)
enddef
```

Exceptions

Fixed point arithmetic

Crossbar Extract

These operations take operands from three registers, perform operations on partitions of bits in the operands, and place the concatenated results in a fourth register.

Operation codes

X.EXTRACT	Crossbar extract

Format

X.EXTRACT ra=rd,rc,rb

ra=xextract(rd,rc,rb)

31..24	23..18	17..12	11..6	5..0
op	rd	rc	rb	ra
8	6	6	6	6

Description

The contents of registers rd, rc, and rb are fetched. The specified operation is performed on these operands. The result is placed into register ra.

Bits 31..0 of the contents of register rb specifies several parameters which control the manner in which data is extracted, and for certain operations, the manner in which the operation is performed. The position of the control fields allows for the source position to be added to a fixed control value for dynamic computation, and allows for the lower 16 bits of the control field to be set for some of the simpler extract cases by a single GCOPYI.128 instruction. The control fields are further arranged so that if only the low order 8 bits are non-zero, a 128-bit extraction with truncation and no rounding is performed.

31..24	23..16	15	14	13	12	11	10..9	8..0
fsize	dpos	x	s	n	m	l	rnd	gssp
8	8	1	1	1	1	1	2	9

The table below describes the meaning of each label:

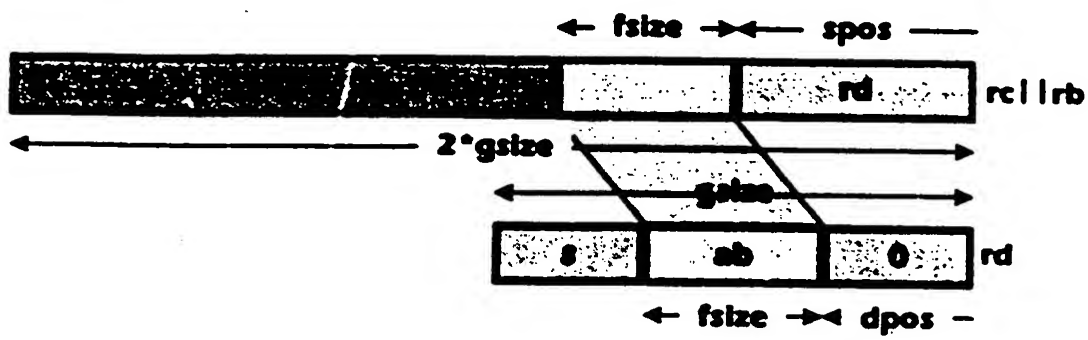
label	bits	meaning
fsize	8	field size
dpos	8	destination position
x	1	reserved
s	1	signed vs. unsigned
n	1	reserved
m	1	merge vs. extract
l	1	reserved
rnd	2	reserved
gssp	9	group size and source position

The 9-bit **gssp** field encodes both the group size, **gsize**, and source position, **spos**, according to the formula **gssp** = 512-4***gsize**+**spos**. The group size, **gsize**, is a power of two in the range 1..128. The source position, **spos**, is in the range 0..(2***gsize**)-1.

The values in the **s**, **n**, **m**, **l**, and **rnd** fields have the following meaning:

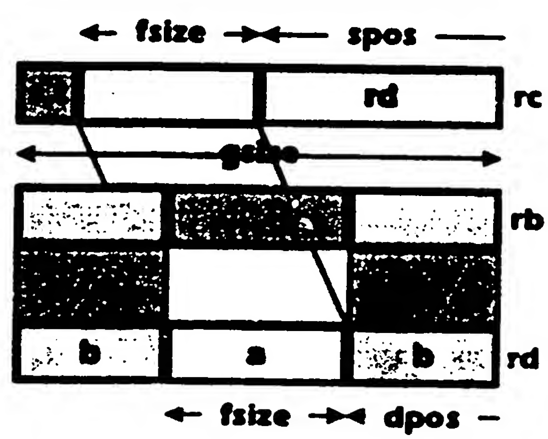
values	s	n	m	l	rnd
0	unsigned		extract		
1	signed		merge		
2					
3					

For the X.EXTRACT instruction, when m=0, the parameters are interpreted to select a fields from the catenated contents of registers rd and rc, extracting values which are catenated and placed in register ra.:



Crossbar extract

For a crossbar-merge-extract (X.EXTRACT when m=1), the parameters are interpreted to merge a fields from the contents of register rd with the contents of register rc. The results are catenated and placed in register ra.



Crossbar merge extract

Definition

```
def CrossbarExtract(op,ra,rb,rc,rd) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    case b8..0 of
        0..255:
            gsize ← 128
        256..383:
            gsize ← 64
        384..447:
            gsize ← 32
        448..479:
            gsize ← 16
        480..495:
            gsize ← 8
        496..503:
            gsize ← 4
        504..507:
            gsize ← 2
        508..511:
            gsize ← 1
    endcase
    m ← b12
    as ← signed ← b14
    h ← (2-m)*gsize
    spos ← (b8..0) and ((2-m)*gsize-1)
    dpos ← (0 || b23..16) and (gsize-1)
    sfsize ← (0 || b31..24) and (gsize-1)
    tfsize ← (sfsize = 0) or ((sfsize+dpos) > gsize) ? gsize-dpos : sfsize
    fsize ← (tfsize + spos > h) ? h - spos : tfsize
    for i ← 0 to 128-gsize by gsize
        case op of
            X.EXTRACT:
                if m then
                    p ← dgsize+i-1..i
                else
                    p ← (d || c)2*(gsize+i)-1..2*i
                endif
        endcase
        v ← (as & ph-1)||p
        w ← (as & vspos+fsize-1)^(gsize-fsize-dpos) || vfsize-1+spos..spos || 0^dpos
        if m then
            asize-1+i..i ← cgsize-1+i..dpos+fsize+i || wdpos+fsize-1..dpos || cdpos-1+i..i
        else
            asize-1+i..i ← w
        endif
    endfor
    RegWrite(ra, 128, a)
enddef
```

Exceptions

none

Crossbar Field

These operations take operands from a register and two immediate values, perform operations on partitions of bits in the operands, and place the concatenated results in the second register.

Operation codes

X.DEPOSIT.2	Crossbar deposit signed pecks
X.DEPOSIT.4	Crossbar deposit signed nibbles
X.DEPOSIT.8	Crossbar deposit signed bytes
X.DEPOSIT.16	Crossbar deposit signed doublets
X.DEPOSIT.32	Crossbar deposit signed quadlets
X.DEPOSIT.64	Crossbar deposit signed octlets
X.DEPOSIT.128	Crossbar deposit signed hexlet
X.DEPOSIT.U.2	Crossbar deposit unsigned pecks
X.DEPOSIT.U.4	Crossbar deposit unsigned nibbles
X.DEPOSIT.U.8	Crossbar deposit unsigned bytes
X.DEPOSIT.U.16	Crossbar deposit unsigned doublets
X.DEPOSIT.U.32	Crossbar deposit unsigned quadlets
X.DEPOSIT.U.64	Crossbar deposit unsigned octlets
X.DEPOSIT.U.128	Crossbar deposit unsigned hexlet
X.WITHDRAW.U.2	Crossbar withdraw unsigned pecks
X.WITHDRAW.U.4	Crossbar withdraw unsigned nibbles
X.WITHDRAW.U.8	Crossbar withdraw unsigned bytes
X.WITHDRAW.U.16	Crossbar withdraw unsigned doublets
X.WITHDRAW.U.32	Crossbar withdraw unsigned quadlets
X.WITHDRAW.U.64	Crossbar withdraw unsigned octlets
X.WITHDRAW.U.128	Crossbar withdraw unsigned hexlet
X.WITHDRAW.2	Crossbar withdraw pecks
X.WITHDRAW.4	Crossbar withdraw nibbles
X.WITHDRAW.8	Crossbar withdraw bytes
X.WITHDRAW.16	Crossbar withdraw doublets
X.WITHDRAW.32	Crossbar withdraw quadlets
X.WITHDRAW.64	Crossbar withdraw octlets
X.WITHDRAW.128	Crossbar withdraw hexlet

Equivalencies

X.SEX.I.2	Crossbar extend immediate signed pecks
X.SEX.I.4	Crossbar extend immediate signed nibbles
X.SEX.I.8	Crossbar extend immediate signed bytes
X.SEX.I.16	Crossbar extend immediate signed doublets
X.SEX.I.32	Crossbar extend immediate signed quadlets
X.SEX.I.64	Crossbar extend immediate signed octlets
X.SEX.I.128	Crossbar extend immediate signed hexlet
X.ZEX.I.2	Crossbar extend immediate unsigned pecks
X.ZEX.I.4	Crossbar extend immediate unsigned nibbles
X.ZEX.I.8	Crossbar extend immediate unsigned bytes
X.ZEX.I.16	Crossbar extend immediate unsigned doublets
X.ZEX.I.32	Crossbar extend immediate unsigned quadlets
X.ZEX.I.64	Crossbar extend immediate unsigned octlets
X.ZEX.I.128	Crossbar extend immediate unsigned hexlet

X.SHL.I.gsize rd=rc,i	→	X.DEPOSIT.gsize rd=rc,size-i,i
X.SHR.I.gsize rd=rc,i	→	X.WITHDRAW.gsize rd=rc,size-i,i
X.SHRU.I.gsize rd=rc,i	→	X.WITHDRAW.U.gsize rd=rc,size-i,i
X.SEX.I.gsize rd=rc,i	→	X.DEPOSIT.gsize rd=rc,i,0
X.ZEX.I.gsize rd=rc,i	→	X.DEPOSIT.U.gsize rd=rc,i,0

Redundancies

X.DEPOSIT.gsize rd=rc,gsize,0	⇔	X.COPY rd=rc
X.DEPOSIT.U.gsize rd=rc,gsize,0	⇔	X.COPY rd=rc
X.WITHDRAW.gsize rd=rc,gsize,0	⇔	X.COPY rd=rc
X.WITHDRAW.U.gsize rd=rc,gsize,0	⇔	X.COPY rd=rc

Format

X.op.gsize		rd=rc,isize,ishift

rd=xopgsize(rc,isize,ishift)

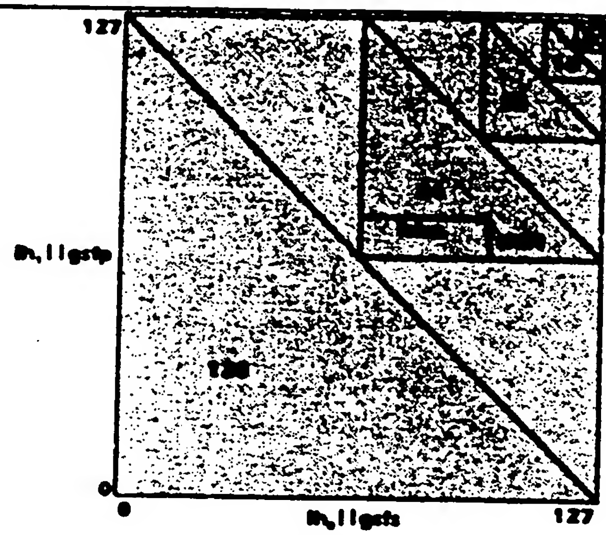
31 26	25 24	23 18	17 12	11 6	5 0
op	ih	rd	rc	gsfp	gsfs
6	2	6	6	6	6

assert isize+ishift ≤ gsize
assert isize≥1
ih_0 || gsfs ← 128-gsize+isize-1
ih_1 || gsfp ← 128-gsize+ishift

Description

The contents of register rc is fetched, and 7-bit immediate values are taken from the 2-bit ih and the 6-bit gsfp and gsfs fields. The specified operation is performed on these operands. The result is placed into register rd.

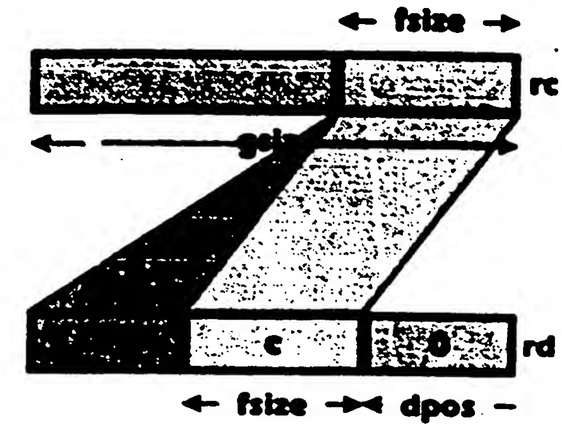
The diagram below shows legal values for the ih, gsfp and gsfs fields, indicating the group size to which they apply.



encoding for crossbar field

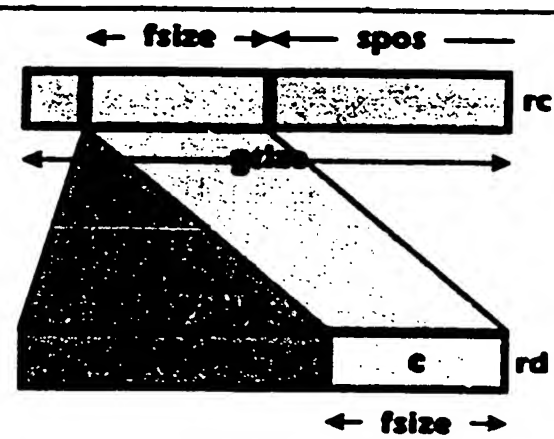
The ih, gsfp and gsfs fields encode three values: the group size, the field size, and a shift amount. The shift amount can also be considered to be the source bit field position for group-withdraw instructions or the destination bit field position for group-deposit instructions. The encoding is designed so that combining the gsfp and gsfs fields with a bitwise-and produces a result which can be decoded to the group size, and so the field size and shift amount can be easily decoded once the group size has been determined.

The crossbar-deposit instructions deposit a bit field from the lower bits of each group partition of the source to a specified bit position in the result. The value is either sign-extended or zero-extended, as specified.



crossbar deposit

The crossbar-withdraw instructions withdraw a bit field from a specified bit position in the each group partition of the source and place it in the lower bits in the result. The value is either sign-extended or zero-extended, as specified.



crossbar withdraw

Definition

```
def CrossbarField(op,rd,rc,gsfp,gsfs) as
    c ← RegRead(rc, 128)
    case ((op1 || gsfp) and (op0 || gsfs)) of
        0..63:
            gsize ← 128
        64..95:
            gsize ← 64
```

96..111:
gsize ← 32
112..119:
gsize ← 16
120..123:
gsize ← 8
124..125:
gsize ← 4
126:
gsize ← 2
127:
raise ReservedInstruction
endcase
ishift ← $(op_1 \mid\mid gsfp)$ and (gsize-1)
isize ← $((op_0 \mid\mid gsfs)$ and (gsize-1))+1
if (ishift+isize>gsize)
raise ReservedInstruction
endif
case op of
X.DEPOSIT:
for i ← 0 to 128-gsize by gsize
$a_{i+gsize-1..i} \leftarrow c_{i+isize-1}^{gsize-isize-ishift} \mid\mid c_{i+isize-1..i} \mid\mid 0^{ishift}$
endfor
X.DEPOSIT.U:
for i ← 0 to 128-gsize by gsize
$a_{i+gsize-1..i} \leftarrow 0^{gsize-isize-ishift} \mid\mid c_{i+isize-1..i} \mid\mid 0^{ishift}$
endfor
X.WITHDRAW:
for i ← 0 to 128-gsize by gsize
$a_{i+gsize-1..i} \leftarrow c_{i+isize+ishift-1}^{size-isize} \mid\mid c_{i+isize+ishift-1..i+ishift}$
endfor
X.WITHDRAW.U:
for i ← 0 to 128-gsize by gsize
$a_{i+gsize-1..i} \leftarrow 0^{gsize-isize} \mid\mid c_{i+isize+ishift-1..i+ishift}$
endfor
endcase
RegWrite(rd, 128, a)
enddef

Exceptions

Reserved instruction

Crossbar Field Inplace

These operations take operands from two registers and two immediate values, perform operations on partitions of bits in the operands, and place the concatenated results in the second register.

Operation codes

X.DEPOSIT.M.2	Crossbar deposit merge pecks
X.DEPOSIT.M.4	Crossbar deposit merge nibbles
X.DEPOSIT.M.8	Crossbar deposit merge bytes
X.DEPOSIT.M.16	Crossbar deposit merge doublets
X.DEPOSIT.M.32	Crossbar deposit merge quadlets
X.DEPOSIT.M.64	Crossbar deposit merge octlets
X.DEPOSIT.M.128	Crossbar deposit merge hexlet

Equivalencies

X.DEPOSIT.M.1	Crossbar deposit merge bits

X.DEPOSIT.M.1 rd@rc,1,0	→	X.COPY rd=rc

Redundancies

X.DEPOSIT.M.gsize rd@rc,gsize,0	⇔	X.COPY rd=rc

Format

X.op.gsize   rd@rc,isize,ishift

rd=xopgsize(rd,rc,isize,ishift)

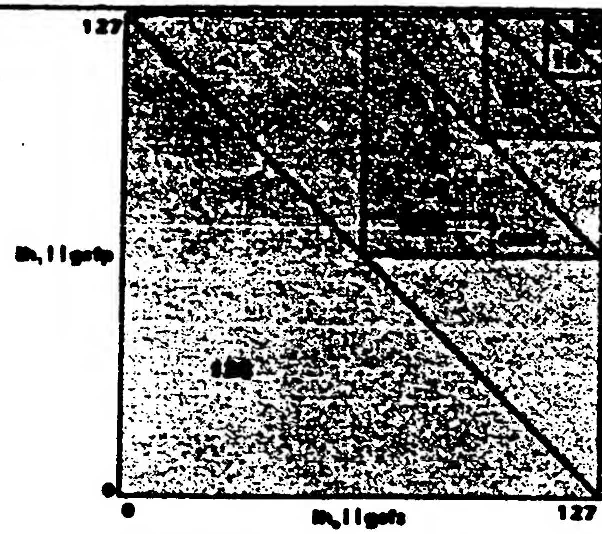
31 26	25 24	23 18	17 12	11 6	5 0
op	ih	rd	rc	gsfp	gsfs
6	2	6	6	6	6

assert $isize+ishift \le gsize$
assert $isize \ge 1$
$ih_0 \,||\, gsfs \leftarrow 128-gsize+isize-1$
$ih_1 \,||\, gsfp \leftarrow 128-gsize+ishift$

Description

The contents of registers rd and rc are fetched, and 7-bit immediate values are taken from the 2-bit ih and the 6-bit gsfp and gsfs fields. The specified operation is performed on these operands. The result is placed into register rd.

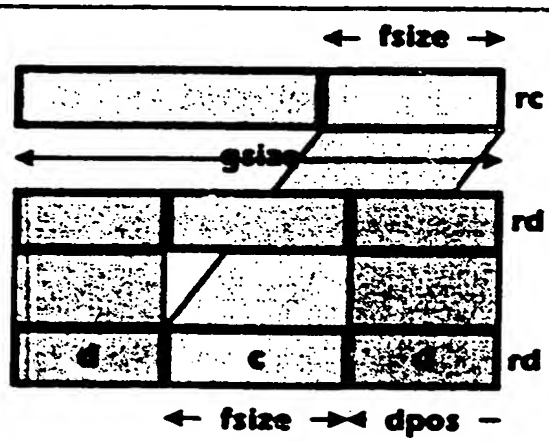
The diagram below shows legal values for the ih, gsfp and gsfs fields, indicating the group size to which they apply.



encoding for crossbar field

The ih, gsfp and gsfs fields encode three values: the group size, the field size, and a shift amount. The shift amount can also be considered to be the source bit field position for group-withdraw instructions or the destination bit field position for group-deposit instructions. The encoding is designed so that combining the gsfp and gsfs fields with a bitwise-and produces a result which can be decoded to the group size, and so the field size and shift amount can be easily decoded once the group size has been determined.

The crossbar-deposit-merge instructions deposit a bit field from the lower bits of each group partition of the source to a specified bit position in the result. The value is merged with the contents of register rd at bit positions above and below the deposited bit field. No sign- or zero-extension is performed by this instruction.



crossbar deposit

Definition

```
def CrossbarFieldInplace(op,rd,rc,gsfp,gsfs) as
    c ← RegRead(rc, 128)
    d ← RegRead(rd, 128)
    case ((op1 || gsfp) and (op0 || gsfs)) of
        0..63:
            gsize ← 128
        64..95:
            gsize ← 64
        96..111:
            gsize ← 32
        112..119:
            gsize ← 16
        120..123:
            gsize ← 8
        124..125:
            gsize ← 4
        126:
            gsize ← 2
        127:
            raise ReservedInstruction
    endcase
    ishift ← (op1 || gsfp) and (gsize-1)
    isize ← ((op0 || gsfs) and (gsize-1))+1
    if (ishift+isize>gsize)
        raise ReservedInstruction
    endif
    for i ← 0 to 128-gsize by gsize
        a_{i+gsize-1..i} ← d_{i+gsize-1..i+isize+ishift} || c_{i+isize-1..i} || d_{i+ishift-1..i}
    endfor
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Reserved instruction

Crossbar Inplace

These operations take operands from three registers, perform operations on partitions of bits in the operands, and place the concatenated results in the third register.

Operation codes

XSHL.M.2	Crossbar shift left merge pecks
XSHL.M.4	Crossbar shift left merge nibbles
XSHL.M.8	Crossbar shift left merge bytes
XSHL.M.16	Crossbar shift left merge doublets
XSHL.M.32	Crossbar shift left merge quadlets
XSHL.M.64	Crossbar shift left merge octlets
XSHL.M.128	Crossbar shift left merge hexlet
XSHR.M.2	Crossbar shift right merge pecks
XSHR.M.4	Crossbar shift right merge nibbles
XSHR.M.8	Crossbar shift right merge bytes
XSHR.M.16	Crossbar shift right merge doublets
XSHR.M.32	Crossbar shift right merge quadlets
XSHR.M.64	Crossbar shift right merge octlets
XSHR.M.128	Crossbar shift right merge hexlet

Format

X.op.size rd@rc,rb

rd=xopsize(rd,rc,rb)

31 25	24	23 18	17 12	11 6	5 2	1 0
XSHIFT	s	rd	rc	rb	op	sz
7	1	6	6	6	4	2

lsize ← log(size)
s ← $lsize_2$
sz ← $lsize_{1..0}$

Description

The contents of registers rd, rc and rb are fetched. The specified operation is performed on these operands. The result is placed into register rd.

Register rd is both a source and destination of this instruction.

Definition

```
def CrossbarInplace(op,size,rd,rc,rb) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
```

```
    b ← RegRead(rb, 128)
    shift ← b and (size-1)
    for i ← 0 to 128-size by size
        case op of
            XSHRM:
                a[i+size-1..i] ← c[i+shift-1..i] || d[i+size-1..i+shift]
            XSHLM:
                a[i+size-1..i] ← d[i+size-1-shift..i] || c[i+shift-1..i]
    endfor
    RegWrite(rd, 128, a)
enddef
```

Exceptions

none

Crossbar Short Immediate

These operations take operands from a register and a short immediate value, perform operations on partitions of bits in the operands, and place the concatenated results in a register.

Operation codes

X.COMPRESS.I.2	Crossbar compress immediate signed pecks
X.COMPRESS.I.4	Crossbar compress immediate signed nibbles
X.COMPRESS.I.8	Crossbar compress immediate signed bytes
X.COMPRESS.I.16	Crossbar compress immediate signed doublets
X.COMPRESS.I.32	Crossbar compress immediate signed quadlets
X.COMPRESS.I.64	Crossbar compress immediate signed octlets
X.COMPRESS.I.128	Crossbar compress immediate signed hexlet
X.COMPRESS.I.U.2	Crossbar compress immediate unsigned pecks
X.COMPRESS.I.U.4	Crossbar compress immediate unsigned nibbles
X.COMPRESS.I.U.8	Crossbar compress immediate unsigned bytes
X.COMPRESS.I.U.16	Crossbar compress immediate unsigned doublets
X.COMPRESS.I.U.32	Crossbar compress immediate unsigned quadlets
X.COMPRESS.I.U.64	Crossbar compress immediate unsigned octlets
X.COMPRESS.I.U.128	Crossbar compress immediate unsigned hexlet
X.EXPAND.I.2	Crossbar expand immediate signed pecks
X.EXPAND.I.4	Crossbar expand immediate signed nibbles
X.EXPAND.I.8	Crossbar expand immediate signed bytes
X.EXPAND.I.16	Crossbar expand immediate signed doublets
X.EXPAND.I.32	Crossbar expand immediate signed quadlets
X.EXPAND.I.64	Crossbar expand immediate signed octlets
X.EXPAND.I.128	Crossbar expand immediate signed hexlet
X.EXPAND.I.U.2	Crossbar expand immediate unsigned pecks
X.EXPAND.I.U.4	Crossbar expand immediate unsigned nibbles
X.EXPAND.I.U.8	Crossbar expand immediate unsigned bytes
X.EXPAND.I.U.16	Crossbar expand immediate unsigned doublets
X.EXPAND.I.U.32	Crossbar expand immediate unsigned quadlets
X.EXPAND.I.U.64	Crossbar expand immediate unsigned octlets
X.EXPAND.I.U.128	Crossbar expand immediate unsigned hexlet
X.ROTL.I.2	Crossbar rotate left immediate pecks
X.ROTL.I.4	Crossbar rotate left immediate nibbles
X.ROTL.I.8	Crossbar rotate left immediate bytes
X.ROTL.I.16	Crossbar rotate left immediate doublets
X.ROTL.I.32	Crossbar rotate left immediate quadlets
X.ROTL.I.64	Crossbar rotate left immediate octlets
X.ROTL.I.128	Crossbar rotate left immediate hexlet
X.ROTR.I.2	Crossbar rotate right immediate pecks
X.ROTR.I.4	Crossbar rotate right immediate nibbles
X.ROTR.I.8	Crossbar rotate right immediate bytes
X.ROTR.I.16	Crossbar rotate right immediate doublets

X.ROTR.I.32	Crossbar rotate right immediate quadlets
X.ROTR.I.64	Crossbar rotate right immediate octlets
X.ROTR.I.128	Crossbar rotate right immediate hexlet
X.SHL.I.2	Crossbar shift left immediate pecks
X.SHL.I.2.O	Crossbar shift left immediate signed pecks check overflow
X.SHL.I.4	Crossbar shift left immediate nibbles
X.SHL.I.4.O	Crossbar shift left immediate signed nibbles check overflow
X.SHL.I.8	Crossbar shift left immediate bytes
X.SHL.I.8.O	Crossbar shift left immediate signed bytes check overflow
X.SHL.I.16	Crossbar shift left immediate doublets
X.SHL.I.16.O	Crossbar shift left immediate signed doublets check overflow
X.SHL.I.32	Crossbar shift left immediate quadlets
X.SHL.I.32.O	Crossbar shift left immediate signed quadlets check overflow
X.SHL.I.64	Crossbar shift left immediate octlets
X.SHL.I.64.O	Crossbar shift left immediate signed octlets check overflow
X.SHL.I.128	Crossbar shift left immediate hexlet
X.SHL.I.128.O	Crossbar shift left immediate signed hexlet check overflow
X.SHL.I.U.2.O	Crossbar shift left immediate unsigned pecks check overflow
X.SHL.I.U.4.O	Crossbar shift left immediate unsigned nibbles check overflow
X.SHL.I.U.8.O	Crossbar shift left immediate unsigned bytes check overflow
X.SHL.I.U.16.O	Crossbar shift left immediate unsigned doublets check overflow
X.SHL.I.U.32.O	Crossbar shift left immediate unsigned quadlets check overflow
X.SHL.I.U.64.O	Crossbar shift left immediate unsigned octlets check overflow
X.SHL.I.U.128.O	Crossbar shift left immediate unsigned hexlet check overflow
X.SHR.I.2	Crossbar signed shift right immediate pecks
X.SHR.I.4	Crossbar signed shift right immediate nibbles
X.SHR.I.8	Crossbar signed shift right immediate bytes
X.SHR.I.16	Crossbar signed shift right immediate doublets
X.SHR.I.32	Crossbar signed shift right immediate quadlets
X.SHR.I.64	Crossbar signed shift right immediate octlets
X.SHR.I.128	Crossbar signed shift right immediate hexlet
X.SHR.I.U.2	Crossbar shift right immediate unsigned pecks
X.SHR.I.U.4	Crossbar shift right immediate unsigned nibbles
X.SHR.I.U.8	Crossbar shift right immediate unsigned bytes
X.SHR.I.U.16	Crossbar shift right immediate unsigned doublets
X.SHR.I.U.32	Crossbar shift right immediate unsigned quadlets
X.SHR.I.U.64	Crossbar shift right immediate unsigned octlets
X.SHR.I.U.128	Crossbar shift right immediate unsigned hexlet

Equivalencies

X.COPY	Crossbar copy
X.NOP	Crossbar no operation

X.COPY rd=rc	←	X.ROTL.I.128 rd=rc,0
X.NOP	←	X.COPY r0=r0

Redundancies

X.ROTL.I.gsize rd=rc,0	⇔	X.COPY rd=rc
X.ROTR.I.gsize rd=rc,0	⇔	X.COPY rd=rc
X.ROTR.I.gsize rd=rc,shift	⇔	X.ROTL.I.gsize rd=rc,gsize-shift
X.SHL.I.gsize rd=rc,0	⇔	X.COPY rd=rc
X.SHL.I.gsize.O rd=rc,0	⇔	X.COPY rd=rc
X.SHL.I.U.gsize.O rd=rc,0	⇔	X.COPY rd=rc
X.SHR.I.gsize rd=rc,0	⇔	X.COPY rd=rc
X.SHR.I.U.gsize rd=rc,0	⇔	X.COPY rd=rc

Selection

class	op	size
precision	COMPRESS.I COMPRESS.I.U EXPAND.I EXPAND.I.U	2 4 8 16 32 64 128
shift	ROTL.I ROTR.I SHL.I SHL.I.O SHL.I.U.O SHR.I SHR.I.U	2 4 8 16 32 64 128
copy	COPY	

Format

X.op.size rd=rc,shift

rd=xopsize(rc,shift)

31..24	23..18	17..12	11..6	5..0
XSHIFTI	rd	rc	simm	op
8	6	6	6	6

$t \leftarrow 256-2^{*}size+shift$
$op_{1..0} \leftarrow t_{7..6}$
$simm \leftarrow t_{5..0}$

Description

A 128-bit value is taken from the contents of register rc. The second operand is taken from simm. The specified operation is performed, and the result is placed in register rd.

Definition

```
def CrossbarShortImmediate(op,rd,rc,simm)
    case (op1..0 || simm) of
        0..127:
            size ← 128
        128..191:
            size ← 64
```

```
        192..223:
            size ← 32
        224..239:
            size ← 16
        240..247:
            size ← 8
        248..251:
            size ← 4
        252..253:
            size ← 2
        254..255:
            raise ReservedInstruction
    endcase
    shift ← (op_0 || simm) and (size-1)
    c ← RegRead(rc, 128)
    case (op_{5..2} || 0^2) of
        X.COMPRESS.I:
            hsize ← size/2
            for i ← 0 to 64-hsize by hsize
                if shift ≤ hsize then
                    a_{i+hsize-1..i} ← c_{i+i+shift+hsize-1..i+i+shift}
                else
                    a_{i+hsize-1..i} ← c_{i+i+size-1}^{shift-hsize} || c_{i+i+size-1..i+i+shift}
                endif
            endfor
            a_{127..64} ← 0
        X.COMPRESS.I.U:
            hsize ← size/2
            for i ← 0 to 64-hsize by hsize
                if shift ≤ hsize then
                    a_{i+hsize-1..i} ← c_{i+i+shift+hsize-1..i+i+shift}
                else
                    a_{i+hsize-1..i} ← 0^{shift-hsize} || c_{i+i+size-1..i+i+shift}
                endif
            endfor
            a_{127..64} ← 0
        X.EXPAND.I:
            hsize ← size/2
            for i ← 0 to 64-hsize by hsize
                if shift ≤ hsize then
                    a_{i+i+size-1..i+i} ← c_{i+hsize-1}^{hsize-shift} || c_{i+hsize-1..i} || 0^{shift}
                else
                    a_{i+i+size-1..i+i} ← c_{i+size-shift-1..i} || 0^{shift}
                endif
            endfor
        X.EXPAND.I.U:
            hsize ← size/2
            for i ← 0 to 64-hsize by hsize
                if shift ≤ hsize then
                    a_{i+i+size-1..i+i} ← 0^{hsize-shift} || c_{i+hsize-1..i} || 0^{shift}
                else
                    a_{i+i+size-1..i+i} ← c_{i+size-shift-1..i} || 0^{shift}
```

endif
endfor
X.SHL.I:
for $i \leftarrow 0$ to 128-size by size
$a_{i+size-1..i} \leftarrow c_{i+size-1-shift..i} \,||\, 0^{shift}$
endfor
X.SHL.I.O:
for $i \leftarrow 0$ to 128-size by size
if $c_{i+size-1..i+size-1-shift} \neq c^{shift+1}_{i+size-1-shift}$ then
raise FixedPointArithmetic
endif
$a_{i+size-1..i} \leftarrow c_{i+size-1-shift..i} \,||\, 0^{shift}$
endfor
X.SHL.I.U.O:
for $i \leftarrow 0$ to 128-size by size
if $c_{i+size-1..i+size-shift} \neq 0^{shift}$ then
raise FixedPointArithmetic
endif
$a_{i+size-1..i} \leftarrow c_{i+size-1-shift..i} \,||\, 0^{shift}$
endfor
X.ROTR.I:
for $i \leftarrow 0$ to 128-size by size
$a_{i+size-1..i} \leftarrow c_{i+shift-1..i} \,||\, c_{i+size-1..i+shift}$
endfor
X.SHR.I:
for $i \leftarrow 0$ to 128-size by size
$a_{i+size-1..i} \leftarrow c^{shift}_{i+size-1} \,||\, c_{i+size-1..i+shift}$
endfor
X.SHR.I.U:
for $i \leftarrow 0$ to 128-size by size
$a_{i+size-1..i} \leftarrow 0^{shift} \,||\, c_{i+size-1..i+shift}$
endfor
endcase
RegWrite(rd, 128, a)
enddef

Exceptions

Fixed point arithmetic
Reserved Instruction

Crossbar Short Immediate Inplace

These operations take operands from two registers and a short immediate value, perform operations on partitions of bits in the operands, and place the concatenated results in the second register.

Operation codes

X.SHL.M.I.2	Crossbar shift left merge immediate pecks
X.SHL.M.I.4	Crossbar shift left merge immediate nibbles
X.SHL.M.I.8	Crossbar shift left merge immediate bytes
X.SHL.M.I.16	Crossbar shift left merge immediate doublets
X.SHL.M.I.32	Crossbar shift left merge immediate quadlets
X.SHL.M.I.64	Crossbar shift left merge immediate octlets
X.SHL.M.I.128	Crossbar shift left merge immediate hexlet
X.SHR.M.I.2	Crossbar shift right merge immediate pecks
X.SHR.M.I.4	Crossbar shift right merge immediate nibbles
X.SHR.M.I.8	Crossbar shift right merge immediate bytes
X.SHR.M.I.16	Crossbar shift right merge immediate doublets
X.SHR.M.I.32	Crossbar shift right merge immediate quadlets
X.SHR.M.I.64	Crossbar shift right merge immediate octlets
X.SHR.M.I.128	Crossbar shift right merge immediate hexlet

Format

X.op.size rd@rc,shift

rd=xopsize(rc,shift)

31 24	23 18	17 12	11 6	5 0
XSHIFTI	rd	rc	simm	op
8	6	6	6	6

$t \leftarrow 256-2^{*}size+shift$
$op_{1..0} \leftarrow t_{7..6}$
$simm \leftarrow t_{5..0}$

Description

Two 128-bit values are taken from the contents of registers rd and rc. A third operand is taken from simm. The specified operation is performed, and the result is placed in register rd.

This instruction is undefined and causes a reserved instruction exception if the simm field is greater or equal to the size specified.

Definition

```
def CrossbarShortImmediateInplace(op,rd,rc,simm)
    case (op1..0 || simm) of
        0..127:
            size ← 128
        128..191:
            size ← 64
        192..223:
            size ← 32
        224..239:
            size ← 16
        240..247:
            size ← 8
        248..251:
            size ← 4
        252..253:
            size ← 2
        254..255:
            raise ReservedInstruction
    endcase
    shift ← (op0 || simm) and (size-1)
    c ← RegRead(rc, 128)
    d ← RegRead(rd, 128)
    for i ← 0 to 128-size by size
        case (op5..2 || 0^2) of
            X.SHR.M.I:
                a(i+size-1..i) ← c(i+shift-1..i) || d(i+size-1..i+shift)
            X.SHL.M.I:
                a(i+size-1..i) ← d(i+size-1-shift..i) || c(i+shift-1..i)
        endcase
    endfor
    RegWrite(rd, 128, a)
enddef
```

Exceptions

Reserved Instruction

Crossbar Shuffle

These operations take operands from two registers, perform operations on partitions of bits in the operands and place the concatenated results in a register.

Operation codes

X.SHUFFLE.4	Crossbar shuffle within pecks
X.SHUFFLE.8	Crossbar shuffle within bytes
X.SHUFFLE.16	Crossbar shuffle within doublets
X.SHUFFLE.32	Crossbar shuffle within quadlets
X.SHUFFLE.64	Crossbar shuffle within octlets
X.SHUFFLE.128	Crossbar shuffle within hexlet
X.SHUFFLE.256	Crossbar shuffle within triclet

Format

X.SHUFFLE.256 rd=rc,rb,v,w,h
X.SHUFFLE.size rd=rcb,v,w

rd=xshuffle256(rc,rb,v,w,h)
rd=xshufflesize(rcb,v,w)

31 24	23 18	17 12	11 6	5 0
X.SHUFFLE	rd	rc	rb	op
8	6	6	6	6

rc ← rb ← rcb

$x \leftarrow \log_2(size)$

$y \leftarrow \log_2(v)$

$z \leftarrow \log_2(w)$

$op \leftarrow ((x*x*x-3*x*x-4*x)/6-(z*z-z)/2+x*z+y) + (size=256)*(h*32-56)$

Description

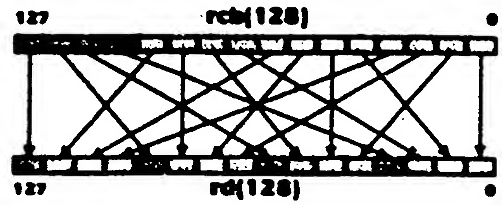
One of two operations are performed depending on whether the **rc** and **rb** fields are equal.

If the **rc** and **rb** fields are equal, a 128-bit operand is taken from the contents of register **rc**. Items of size **v** are divided into **w** piles and shuffled together, within groups of **size** bits, according to the value of **op**. The result is placed in register **rd**.

If the **rc** and **rb** fields are not equal, the contents of registers **rc** and **rb** are catenated into a 256-bit operand. Items of size **v** are divided into **w** piles and shuffled together, according to the value of **op**. Depending on the value of **h**, a sub-field of **op**, the low 128 bits (h=0), or the high 128 bits (h=1) of the 256-bit shuffled contents are selected as the result. The result is placed in register **rd**.

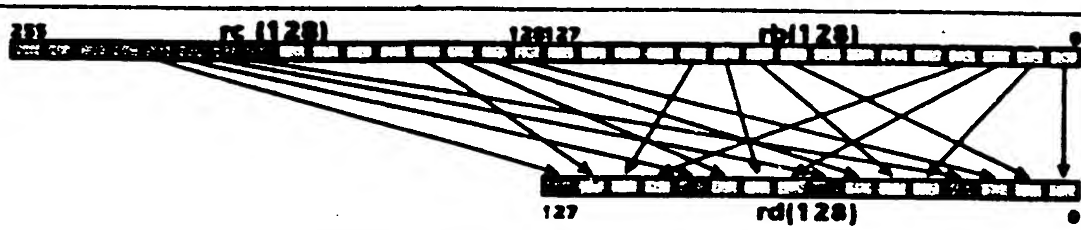
This instruction is undefined and causes a reserved instruction exception if rc and rb are not equal and the op field is greater or equal to 56, or if rc and rb are equal and $op_{4..0}$ is greater or equal to 28.

A crossbar 4-way shuffle of bytes within hexlet instruction (X.SHUFFLE.128 rd=rcb,8,4) divides the 128-bit operand into 16 bytes and partitions the bytes 4 ways (indicated by varying shade in the diagram below). The 4 partitions are perfectly shuffled, producing a 128-bit result.



4-way shuffle bytes within hexlet

A crossbar 4-way shuffle of bytes within triclet instruction (X.SHUFFLE.256 rd=rc,rb,8,4,0) catenates the contents of rc and rb, then divides the 256-bit content into 32 bytes and partitions the bytes 4 ways (indicated by varying shade in the diagram below). The low-order halves of the 4 partitions are perfectly shuffled, producing a 128-bit result.



4-way shuffle bytes within triclet

Changing the last immediate value h to 1 (X.SHUFFLE.256 rd=rc,rb,8,4,1) modifies the operation to perform the same function on the high-order halves of the 4 partitions.

When rc and rb are equal, the table below shows the value of the **op** field and associated values for **size**, **v**, and **w**.

op	size	v	w
0	4	1	2
1	8	1	2
2	8	2	2
3	8	1	4
4	16	1	2
5	16	2	2
6	16	4	2
7	16	1	4
8	16	2	4
9	16	1	8
10	32	1	2
11	32	2	2
12	32	4	2
13	32	8	2
14	32	1	4
15	32	2	4
16	32	4	4
17	32	1	8
18	32	2	8
19	32	1	16
20	64	1	2
21	64	2	2
22	64	4	2
23	64	8	2
24	64	16	2
25	64	1	4
26	64	2	4
27	64	4	4
28	64	8	4
29	64	1	8
30	64	2	8
31	64	4	8
32	64	1	16
33	64	2	16
34	64	1	32
35	128	1	2
36	128	2	2
37	128	4	2
38	128	8	2
39	128	16	2
40	128	32	2
41	128	1	4
42	128	2	4
43	128	4	4
44	128	8	4
45	128	16	4
46	128	1	8
47	128	2	8
48	128	4	8
49	128	8	8
50	128	1	16
51	128	2	16
52	128	4	16
53	128	1	32
54	128	2	32
55	128	1	64

When rc and rb are not equal, the table below shows the value of the $op_{4..0}$ field and associated values for size, v, and w: Op_5 is the value of h, which controls whether the low-order or high-order half of each partition is shuffled into the result.

op4..0	size	v	w
0	256	1	2
1	256	2	2
2	256	4	2
3	256	8	2
4	256	16	2
5	256	32	2
6	256	64	2
7	256	1	4
8	256	2	4
9	256	4	4
10	256	8	4
11	256	16	4
12	256	32	4
13	256	1	8
14	256	2	8
15	256	4	8
16	256	8	8
17	256	16	8
18	256	1	16
19	256	2	16
20	256	4	16
21	256	8	16
22	256	1	32
23	256	2	32
24	256	4	32
25	256	1	64
26	256	2	64
27	256	1	128

Definition

```
def CrossbarShuffle(major,rd,rc,rb,op)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    if rc=rb then
        case op of
            0..55:
                for x ← 2 to 7; for y ← 0 to x-2; for z ← 1 to x-y-1
                    if op = ((x*x*x-3*x*x-4*x)/6-(z*z-z)/2+x*z+y) then
                        for i ← 0 to 127
                            ai ← c(i6..x || iy+z-1..y || ix-1..y+z || iy-1..0)
                        end
                    endif
                endfor; endfor; endfor
            56..63:
                raise ReservedInstruction
```

```
            endcase
        elseif
            case op4..0 of
                0..27:
                    cb ← c || b
                    x ← 8
                    h ← op5
                    for y ← 0 to x-2; for z ← 1 to x-y-1
                        if op4..0 = ((17*z-z*z)/2-8+y) then
                            for i ← h*128 to 127+h*128
                                ai-h*128 ← cb(iy+z-1..y || ix-1..y+z || iy-1..0)
                            end
                        endif
                    endfor; endfor
                28..31:
                    raise ReservedInstruction
            endcase
        endif
        RegWrite(rd, 128, a)
enddef
```

Exceptions

Reserved Instruction

Crossbar Swizzle

These operations perform calculations with a general register value and immediate values, placing the result in a general register.

Operation codes

X.SWIZZLE	Crossbar swizzle

Format

X.SWIZZLE rd=rc,icopy,iswap

rd=xswizzle(rc,icopy,iswap)

31 26	25 24	23 18	17 12	11 6	5 0
X.SWIZZLE	ih	rd	rc	icopya	iswapa
6	2	6	6	6	6

$icopya \leftarrow icopy_{5..0}$
$iswapa \leftarrow iswap_{5..0}$
$ih \leftarrow icopy_6 \mid\mid iswap_6$

Description

The contents of register rc are fetched, and 7-bit immediate values, icopy and iswap, are constructed from the 2-bit ih field and from the 6-bit icopya and iswapa fields. The specified operation is performed on these operands. The result is placed into register rd.

Definition

```
def GroupSwizzleImmediate(ih,rd,rc,icopya,iswapa) as
    icopy ← ih1 || icopya
    iswap ← ih0 || iswapa
    c ← RegRead(rc, 128)
    for i ← 0 to 127
        ai ← c(i & icopy) ^ iswap
    endfor
    RegWrite(rd, 128, a)
enddef
```

Exceptions

none

Crossbar Ternary

These operations take three values from registers, perform a group of calculations on partitions of bits of the operands and place the catenated results in a fourth register.

Operation codes

XSELECT.8	Crossbar select bytes

Format

op ra=rd,rc,rb

ra=op(rd,rc,rb)

31 24	23 18	17 12	11 6	5 0
op	rd	rc	rb	ra
8	6	6	6	6

Description

The contents of registers rd, rc, and rb are fetched. The specified operation is performed on these operands. The result is placed into register ra.

Definition

```
def CrossbarTernary(op,rd,rc,rb,ra) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    dc ← d || c
    for i ← 0 to 15
        j ← b8*i+4..8*i
        a8*i+7..8*i ← dc8*j+7..8*j
    endfor
    RegWrite(ra, 128, a)
enddef
```

Exceptions

none

Ensemble

These operations take operands from two registers, perform operations on partitions of bits in the operands, and place the concatenated results in a third register.

Operation codes

E.CON.8	Ensemble convolve signed bytes
E.CON.16	Ensemble convolve signed doublets
E.CON.32	Ensemble convolve signed quadlets
E.CON.64	Ensemble convolve signed octlets
E.CON.C.8	Ensemble convolve complex bytes
E.CON.C.16	Ensemble convolve complex doublets
E.CON.C.32	Ensemble convolve complex quadlets
E.CON.M.8	Ensemble convolve mixed-signed bytes
E.CON.M.16	Ensemble convolve mixed-signed doublets
E.CON.M.32	Ensemble convolve mixed-signed quadlets
E.CON.M.64	Ensemble convolve mixed-signed octlets
E.CON.U.8	Ensemble convolve unsigned bytes
E.CON.U.16	Ensemble convolve unsigned doublets
E.CON.U.32	Ensemble convolve unsigned quadlets
E.CON.U.64	Ensemble convolve unsigned octlets
E.DIV.64	Ensemble divide signed octlets
E.DIV.U.64	Ensemble divide unsigned octlets
E.MUL.8	Ensemble multiply signed bytes
E.MUL.16	Ensemble multiply signed doublets
E.MUL.32	Ensemble multiply signed quadlets
E.MUL.64	Ensemble multiply signed octlets
E.MUL.SUM.8	Ensemble multiply sum signed bytes
E.MUL.SUM.16	Ensemble multiply sum signed doublets
E.MUL.SUM.32	Ensemble multiply sum signed quadlets
E.MUL.SUM.64	Ensemble multiply sum signed octlets
E.MUL.C.8	Ensemble complex multiply bytes
E.MUL.C.16	Ensemble complex multiply doublets
E.MUL.C.32	Ensemble complex multiply quadlets
E.MUL.M.8	Ensemble multiply mixed-signed bytes
E.MUL.M.16	Ensemble multiply mixed-signed doublets
E.MUL.M.32	Ensemble multiply mixed-signed quadlets
E.MUL.M.64	Ensemble multiply mixed-signed octlets
E.MUL.P.8	Ensemble multiply polynomial bytes
E.MUL.P.16	Ensemble multiply polynomial doublets
E.MUL.P.32	Ensemble multiply polynomial quadlets
E.MUL.P.64	Ensemble multiply polynomial octlets
E.MUL.SUM.C.8	Ensemble multiply sum complex bytes
E.MUL.SUM.C.16	Ensemble multiply sum complex doublets
E.MUL.SUM.C.32	Ensemble multiply sum complex quadlets

E.MULSUM.M.8	Ensemble multiply sum mixed-signed bytes
E.MULSUM.M.16	Ensemble multiply sum mixed-signed doublets
E.MULSUM.M.32	Ensemble multiply sum mixed-signed quadlets
E.MULSUM.M.64	Ensemble multiply sum mixed-signed octlets
E.MULSUM.U.8	Ensemble multiply sum unsigned bytes
E.MULSUM.U.16	Ensemble multiply sum unsigned doublets
E.MULSUM.U.32	Ensemble multiply sum unsigned quadlets
E.MULSUM.U.64	Ensemble multiply sum unsigned octlets
E.MUL.U.8	Ensemble multiply unsigned bytes
E.MUL.U.16	Ensemble multiply unsigned doublets
E.MUL.U.32	Ensemble multiply unsigned quadlets
E.MUL.U.64	Ensemble multiply unsigned octlets

Selection

class	op	type	size
multiply	E.MUL	NONE M U P	8 16 32 64
		C	8 16 32
multiply sum	E.MULSUM	NONE M U	8 16 32 64
		C	8 16 32
divide	E.DIV	NONE U	64

Format

E.op.size rd=rc,rb

rd=eopsize(rc,rb)

31 … 24	23 … 18	17 … 12	11 … 6	5 … 0
E.size	rd	rc	rb	op
8	6	6	6	6

Description

Two values are taken from the contents of registers rc and rb. The specified operation is performed, and the result is placed in register rd.

Definition

def mul(size,h,vs,v,i,ws,w,j) as

$\quad mul \leftarrow ((vs \& v_{size-1+i})^{h-size} \mid\mid v_{size-1+i..i}) * ((ws \& w_{size-1+j})^{h-size} \mid\mid w_{size-1+j..j})$

enddef

def c ← PolyMultiply(size,a,b) as

$\quad p[0] \leftarrow 0^{2*size}$

$\quad$ for k ← 0 to size-1

$\quad\quad p[k+1] \leftarrow p[k] \wedge a_k \mathbin{?} (0^{size-k} \mid\mid b \mid\mid 0^k) : 0^{2*size}$

$\quad$ endfor

$\quad c \leftarrow p[size]$

```
enddef

def Ensemble(op,size,rd,rc,rb)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    case op of
        E.MUL:, E.MUL.C:, E.MUL.SUM, E.MUL.SUM.C, E.CON, E.CON.C, E.DIV:
            cs ← bs ← 1
        E.MUL.M:, E.MUL.SUM.M, E.CON.M:
            cs ← 0
            bs ← 1
        E.MUL.U:, E.MUL.SUM.U, E.CON.U, E.DIV.U, E.MUL.P:
            cs ← bs ← 0
    endcase
    case op of
        E.MUL, E.MUL.U, E.MUL.M:
            for i ← 0 to 64-size by size
                d2*(i+size)-1..2*i ← mul(size,2*size,cs,c,i,bs,b,i)
            endfor
        E.MUL.P:
            for i ← 0 to 64-size by size
                d2*(i+size)-1..2*i ← PolyMultiply(size,csize-1+i..i,bsize-1+i..i)
            endfor
        E.MUL.C:
            for i ← 0 to 64-size by size
                if (i and size) = 0 then
                    p ← mul(size,2*size,1,c,i,1,b,i) - mul(size,2*size,1,c,i+size,1,b,i+size)
                else
                    p ← mul(size,2*size,1,c,i,1,b,i+size) + mul(size,2*size,1,c,i,1,b,i+size)
                endif
                d2*(i+size)-1..2*i ← p
            endfor
        E.MUL.SUM, E.MUL.SUM.U, E.MUL.SUM.M:
            p[0] ← 0^128
            for i ← 0 to 128-size by size
                p[i+size] ← p[i] + mul(size,128,cs,c,i,bs,b,i)
            endfor
            a ← p[128]
        E.MUL.SUM.C:
            p[0] ← 0^64
            p[size] ← 0^64
            for i ← 0 to 128-size by size
                if (i and size) = 0 then
                    p[i+2*size] ← p[i] + mul(size,64,1,c,i,1,b,i)
                                       - mul(size,64,1,c,i+size,1,b,i+size)
                else
                    p[i+2*size] ← p[i] + mul(size,64,1,c,i,1,b,i+size)
                                       + mul(size,64,1,c,i+size,1,b,i)
                endif
            endfor
            a ← p[128+size] || p[128]
        E.CON, E.CON.U, E.CON.M:
            p[0] ← 0^128
            for j ← 0 to 64-size by size
```

```
                for i ← 0 to 64-size by size
                    p[j+size]_{2*(i+size)-1..2*i} ← p[j]_{2*(i+size)-1..2*i} +
                        mul(size,2*size,cs,c,i+64-j,bs,b,j)
                endfor
            endfor
            a ← p[64]
        E.CON.C:
            p[0] ← 0^128
            for j ← 0 to 64-size by size
                for i ← 0 to 64-size by size
                    if ((~i) and j and size) = 0 then
                        p[j+size]_{2*(i+size)-1..2*i} ← p[j]_{2*(i+size)-1..2*i} +
                            mul(size,2*size,1,c,i+64-j,1,b,j)
                    else
                        p[j+size]_{2*(i+size)-1..2*i} ← p[j]_{2*(i+size)-1..2*i} -
                            mul(size,2*size,1,c,i+64-j+2*size,1,b,j)
                    endif
                endfor
            endfor
            a ← p[64]
        E.DIV:
            if (b = 0) or ( (c = (1||0^63)) and (b = 1^64) ) then
                a ← undefined
            else
                q ← c / b
                r ← c - q*b
                a ← r_{63..0} || q_{63..0}
            endif
        E.DIV.U:
            if b = 0 then
                a ← undefined
            else
                q ← (0 || c) / (0 || b)
                r ← c - (0 || q)*(0 || b)
                a ← r_{63..0} || q_{63..0}
            endif
    endcase
    RegWrite(rd, 128, a)
enddef
```

Exceptions

none

Ensemble Convolve Extract Immediate

These instructions take an address from a general register to fetch a large operand from memory, a second operand from a general register, perform a group of operations on partitions of bits in the operands, and catenate the results together, placing the result in a general register .

Operation codes

E.CON.X.I.C.8.C.B	Ensemble convolve extract immediate signed complex bytes big-endian ceiling
E.CON.X.I.C.8.F.B	Ensemble convolve extract immediate signed complex bytes big-endian floor
E.CON.X.I.C.8.N.B	Ensemble convolve extract immediate signed complex bytes big-endian nearest
E.CON.X.I.C.8.Z.B	Ensemble convolve extract immediate signed complex bytes big-endian zero
E.CON.X.I.C.16.C.B	Ensemble convolve extract immediate signed complex doublets big-endian ceiling
E.CON.X.I.C.16.F.B	Ensemble convolve extract immediate signed complex doublets big-endian floor
E.CON.X.I.C.16.N.B	Ensemble convolve extract immediate signed complex doublets big-endian nearest
E.CON.X.I.C.16.Z.B	Ensemble convolve extract immediate signed complex doublets big-endian zero
E.CON.X.I.C.32.C.B	Ensemble convolve extract immediate signed complex quadlets big-endian ceiling
E.CON.X.I.C.32.F.B	Ensemble convolve extract immediate signed complex quadlets big-endian floor
E.CON.X.I.C.32.N.B	Ensemble convolve extract immediate signed complex quadlets big-endian nearest
E.CON.X.I.C.32.Z.B	Ensemble convolve extract immediate signed complex quadlets big-endian zero
E.CON.X.I.C.64.C.B	Ensemble convolve extract immediate signed complex octlets big-endian ceiling
E.CON.X.I.C.64.F.B	Ensemble convolve extract immediate signed complex octlets big-endian floor
E.CON.X.I.C.64.N.B	Ensemble convolve extract immediate signed complex octlets big-endian nearest
E.CON.X.I.C.64.Z.B	Ensemble convolve extract immediate signed complex octlets big-endian zero
E.CON.X.I.C.8.C.L	Ensemble convolve extract immediate signed complex bytes little-endian ceiling
E.CON.X.I.C.8.F.L	Ensemble convolve extract immediate signed complex bytes little-endian floor
E.CON.X.I.C.8.N.L	Ensemble convolve extract immediate signed complex bytes little-endian nearest
E.CON.X.I.C.8.Z.L	Ensemble convolve extract immediate signed complex bytes little-endian zero
E.CON.X.I.C.16.C.L	Ensemble convolve extract immediate signed complex doublets little-endian ceiling
E.CON.X.I.C.16.F.L	Ensemble convolve extract immediate signed complex doublets little-endian floor
E.CON.X.I.C.16.N.L	Ensemble convolve extract immediate signed complex doublets little-endian nearest
E.CON.X.I.C.16.Z.L	Ensemble convolve extract immediate signed complex doublets little-endian zero
E.CON.X.I.C.32.C.L	Ensemble convolve extract immediate signed complex quadlets little-endian ceiling
E.CON.X.I.C.32.F.L	Ensemble convolve extract immediate signed complex quadlets little-endian floor
E.CON.X.I.C.32.N.L	Ensemble convolve extract immediate signed complex quadlets little-endian nearest
E.CON.X.I.C.32.Z.L	Ensemble convolve extract immediate signed complex quadlets little-endian zero
E.CON.X.I.C.64.C.L	Ensemble convolve extract immediate signed complex octlets little-endian ceiling
E.CON.X.I.C.64.F.L	Ensemble convolve extract immediate signed complex octlets little-endian floor
E.CON.X.I.C.64.N.L	Ensemble convolve extract immediate signed complex octlets little-endian nearest
E.CON.X.I.C.64.Z.L	Ensemble convolve extract immediate signed complex octlets little-endian zero
E.CON.X.I.8.C.B	Ensemble convolve extract immediate signed bytes big-endian ceiling
E.CON.X.I.8.F.B	Ensemble convolve extract immediate signed bytes big-endian floor
E.CON.X.I.8.N.B	Ensemble convolve extract immediate signed bytes big-endian nearest
E.CON.X.I.8.Z.B	Ensemble convolve extract immediate signed bytes big-endian zero
E.CON.X.I.16.C.B	Ensemble convolve extract immediate signed doublets big-endian ceiling

E.CONX.I.16.F.B	Ensemble convolve extract immediate signed doublets big-endian floor
E.CONX.I.16.N.B	Ensemble convolve extract immediate signed doublets big-endian nearest
E.CONX.I.16.Z.B	Ensemble convolve extract immediate signed doublets big-endian zero
E.CONX.I.32.C.B	Ensemble convolve extract immediate signed quadlets big-endian ceiling
E.CONX.I.32.F.B	Ensemble convolve extract immediate signed quadlets big-endian floor
E.CONX.I.32.N.B	Ensemble convolve extract immediate signed quadlets big-endian nearest
E.CONX.I.32.Z.B	Ensemble convolve extract immediate signed quadlets big-endian zero
E.CONX.I.64.C.B	Ensemble convolve extract immediate signed octlets big-endian ceiling
E.CONX.I.64.F.B	Ensemble convolve extract immediate signed octlets big-endian floor
E.CONX.I.64.N.B	Ensemble convolve extract immediate signed octlets big-endian nearest
E.CONX.I.64.Z.B	Ensemble convolve extract immediate signed octlets big-endian zero
E.CONX.I.8.C.L	Ensemble convolve extract immediate signed bytes little-endian ceiling
E.CONX.I.8.F.L	Ensemble convolve extract immediate signed bytes little-endian floor
E.CONX.I.8.N.L	Ensemble convolve extract immediate signed bytes little-endian nearest
E.CONX.I.8.Z.L	Ensemble convolve extract immediate signed bytes little-endian zero
E.CONX.I.16.C.L	Ensemble convolve extract immediate signed doublets little-endian ceiling
E.CONX.I.16.F.L	Ensemble convolve extract immediate signed doublets little-endian floor
E.CONX.I.16.N.L	Ensemble convolve extract immediate signed doublets little-endian nearest
E.CONX.I.16.Z.L	Ensemble convolve extract immediate signed doublets little-endian zero
E.CONX.I.32.C.L	Ensemble convolve extract immediate signed quadlets little-endian ceiling
E.CONX.I.32.F.L	Ensemble convolve extract immediate signed quadlets little-endian floor
E.CONX.I.32.N.L	Ensemble convolve extract immediate signed quadlets little-endian nearest
E.CONX.I.32.Z.L	Ensemble convolve extract immediate signed quadlets little-endian zero
E.CONX.I.64.C.L	Ensemble convolve extract immediate signed octlets little-endian ceiling
E.CONX.I.64.F.L	Ensemble convolve extract immediate signed octlets little-endian floor
E.CONX.I.64.N.L	Ensemble convolve extract immediate signed octlets little-endian nearest
E.CONX.I.64.Z.L	Ensemble convolve extract immediate signed octlets little-endian zero
E.CONX.I.M.8.C.B	Ensemble convolve extract immediate mixed signed bytes big-endian ceiling
E.CONX.I.M.8.F.B	Ensemble convolve extract immediate mixed signed bytes big-endian floor
E.CONX.I.M.8.N.B	Ensemble convolve extract immediate mixed signed bytes big-endian nearest
E.CONX.I.M.8.Z.B	Ensemble convolve extract immediate mixed signed bytes big-endian zero
E.CONX.I.M.16.C.B	Ensemble convolve extract immediate mixed-signed doublets big-endian ceiling
E.CONX.I.M.16.F.B	Ensemble convolve extract immediate mixed-signed doublets big-endian floor
E.CONX.I.M.16.N.B	Ensemble convolve extract immediate mixed-signed doublets big-endian nearest
E.CONX.I.M.16.Z.B	Ensemble convolve extract immediate mixed-signed doublets big-endian zero
E.CONX.I.M.32.C.B	Ensemble convolve extract immediate mixed signed quadlets big-endian ceiling
E.CONX.I.M.32.F.B	Ensemble convolve extract immediate mixed-signed quadlets big-endian floor
E.CONX.I.M.32.N.B	Ensemble convolve extract immediate mixed signed quadlets big-endian nearest
E.CONX.I.M.32.Z.B	Ensemble convolve extract immediate mixed signed quadlets big-endian zero
E.CONX.I.M.64.C.B	Ensemble convolve extract immediate mixed-signed octlets big-endian ceiling
E.CONX.I.M.64.F.B	Ensemble convolve extract immediate mixed-signed octlets big-endian floor
E.CONX.I.M.64.N.B	Ensemble convolve extract immediate mixed-signed octlets big-endian nearest
E.CONX.I.M.64.Z.B	Ensemble convolve extract immediate mixed-signed octlets big-endian zero
E.CONX.I.M.8.C.L	Ensemble convolve extract immediate mixed signed bytes little-endian ceiling
E.CONX.I.M.8.F.L	Ensemble convolve extract immediate mixed-signed bytes little-endian floor
E.CONX.I.M.8.N.L	Ensemble convolve extract immediate mixed signed bytes little-endian nearest

E.CON.X.I.M.8.Z.L	Ensemble convolve extract immediate mixed signed bytes little endian zero
E.CON.X.I.M.16.C.L	Ensemble convolve extract immediate mixed signed doublets little endian ceiling
E.CON.X.I.M.16.F.L	Ensemble convolve extract immediate mixed signed doublets little endian floor
E.CON.X.I.M.16.N.L	Ensemble convolve extract immediate mixed signed doublets little endian nearest
E.CON.X.I.M.16.Z.L	Ensemble convolve extract immediate mixed signed doublets little endian zero
E.CON.X.I.M.32.C.L	Ensemble convolve extract immediate mixed signed quadlets little endian ceiling
E.CON.X.I.M.32.F.L	Ensemble convolve extract immediate mixed signed quadlets little endian floor
E.CON.X.I.M.32.N.L	Ensemble convolve extract immediate mixed sig[illegible] quadlets little endian nearest
E.CON.X.I.M.32.Z.L	Ensemble convolve extract immediate mixed signed quadlets little endian zero
E.CON.X.I.M.64.C.L	Ensemble [illegible] extract immediate mixed signed octlets little endian ceiling
E.CON.X.I.M.64.F.L	Ensemble convolve extract immediate mixed signed octlets little endian floor
E.CON.X.I.M.64.N.L	Ensemble convolve extract immediate mixed signed octlets little endian nearest
E.CON.X.I.M.64.Z.L	Ensemble [illegible]onvolve extract immediate mixed signed octlets little endian zero
E.CON.X.I.U.8.C.B	Ensemble convolve extract immediate unsigned bytes big endian ceiling
E.CON.X.I.U.8.F.B	Ensemble convolve extract immediate unsigned bytes big endian floor
E.CON.X.I.U.8.N.B	Ensemble convolve extract immediate unsigned bytes big endian nearest
E.CON.X.I.U.16.C.B	Ensemble convolve extract immediate unsigned doublets big endian ceiling
E.CON.X.I.U.16.F.B	Ensemble convolve extract immediate unsigned doublets big endian floor
E.CON.X.I.U.16.N.B	Ensemble convolve extract immediate unsigned doublets big endian nearest
E.CON.X.I.U.32.C.B	Ensemble convolve extract immediate unsigned quadlets big endian ceiling
E.CON.X.I.U.32.F.B	Ensemble convolve extract immediate unsigned quadlets big endian floor
E.CON.X.I.U.32.N.B	Ensemble convolve extract immediate unsigned quadlets big endian nearest
E.CON.X.I.U.64.C.B	Ensemble convolve extract immediate unsigned octlets big-endian ceiling
E.CON.X.I.U.64.F.B	Ensemble convolve extract immediate unsigned octlets big-endian floor
E.CON.X.I.U.64.N.B	Ensemble convolve extract immediate unsigned octlets big-endian nearest
E.CON.X.I.U.8.C.L	Ensemble convolve extract immediate unsigned bytes little endian ceiling
E.CON.X.I.U.8.F.L	Ensemble convolve extract immediate unsigned bytes little endian floor
E.CON.X.I.U.8.N.L	Ensemble convolve extract immediate unsigned bytes little endian nearest
E.CON.X.I.U.16.C.L	Ensemble convolve extract immediate unsigned doublets little endian ceiling
E.CON.X.I.U.16.F.L	Ensemble convolve extract immediate unsigned doublets little endian floor
E.CON.X.I.U.16.N.L	Ensemble convolve extract immediate unsigned doublets little endian nearest
E.CON.X.I.U.32.C.L	Ensemble convolve extra[illegible] immediate unsigned quadlets little endian ceiling
E.CON.X.I.U.32.F.L	Ensemble convolve extract immediate unsigned quadlets little endian floor
E.CON.X.I.U.32.N.L	Ensemble convolve extract immediate unsigned quadlets little endian nearest
E.CON.X.I.U.64.C.L	Ensemble convolve extract immediate unsigned octlets little endian ceiling
E.CON.X.I.U.64.F.L	Ensemble convolve extract immediate unsigned octlets little endian floor
E.CON.X.I.U.64.N.L	Ensemble convolve extract immediate unsigned octlets little endian nearest

Format

E.op.size.rnd rd@rc,rb,i

rd=eopsizernd(rd,rc,rb,i)

31 … 24	23 … 18	17 … 12	11 … 6	5 4	3 2	1 0
E.op	rd	rc	rb	sz	rnd	sh
8	6	6	6	2	2	2

$sz \leftarrow \log(size) - 3$
$sh \leftarrow size + 7 - \log(size) - i$

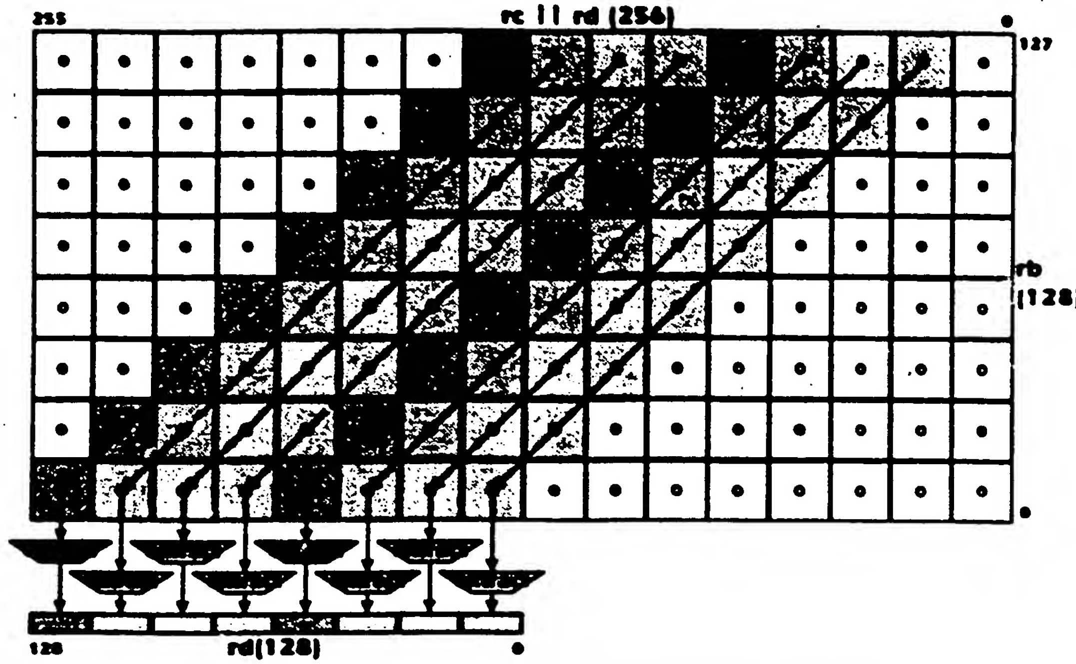
Description

The contents of registers rd and rc are catenated, as specified by the order parameter, and used as a first value. A second value is the contents of register rb. The values are partitioned into groups of operands of the size specified and are convolved, producing a group of values. The group of values is rounded, and limited as specified, yielding a group of results which is the size specified. The group of results is catenated and placed in register rd.

Z (zero) rounding is not defined for unsigned extract operations, and a ReservedInstruction exception is raised if attempted. F (floor) rounding will properly round unsigned results downward.

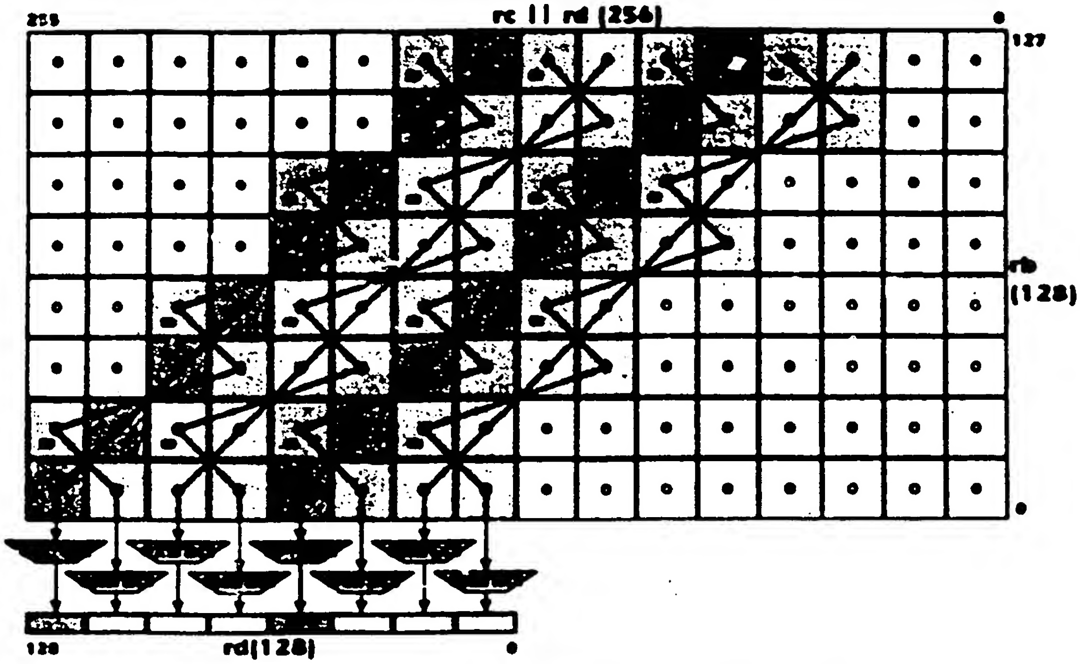
The order parameter of the instruction specifies the order in which the contents of registers rd and rc are catenated. The choice is significant because the contents of register rd is overwritten. When little-endian order is specified, the contents are catenated so that the contents of register rc is most significant (left) and the contents of register rd is least significant (right). When big-endian order is specified, the contents are catenated so that the contents of register rd is most significant (left) and the contents of register rc is least significant (right).

An ensemble-convolve-extract-immediate-doublets instruction (E.CON.X.I16, E.CON.X.IM16, or E.CON.X.IU16) convolves vector [x w v u t s r q p o n m l k j i] with vector [h g f e d c b a], yielding the products [ax+bw+cv+du+et+fs+gr+hq ... as+br+cq+dp+eo+fn+gm+hl ar+bq+cp+do+en+fm+gl+hk aq+bp+co+dn+em+fl+gk+hj], rounded and limited as specified:



Ensemble convolve extract immediate doublets

An ensemble-convolve-extract-immediate-complex-doublets instruction (E.CON.X.IC16) convolves vector [x w v u t s r q p o n m l k j i] with vector [h g f e d c b a], yielding the products [ax+bw+cv+du+et+fs+gr+hq ... as-bt+cq-dr+eo-fp+gm-hn ar+bq+cp+do+en-fm+gl+hk aq-br+co-dp+em-fn+gk+hl], rounded and limited as specified.



Ensemble convolve extract immediate complex doublets

Definition

```
def mul(size,h,vs,v,i,ws,w,j) as
      mul ← ((vs&v_size-1+i)^(h-size) || v_size-1+i..i) * ((ws&w_size-1+j)^(h-size) || w_size-1+j..j)
enddef

def EnsembleConvolveExtractImmediate(op,rnd,gsize,rd,rc,rb,sh)
      d ← RegRead(rd, 128)
      c ← RegRead(rd, 128)
      b ← RegRead(rb, 128)
      lgsize ← log(gsize)
      wsize ← 128
      msize ← 256
      vsize ← 128
      case op of
            E.CON.X.I.B, E.CON.X.I.U.B, E.CON.X.I.M.B, E.CON.X.I.C.B:
                  m ← d || c
            E.CON.X.I.L, E.CON.X.I.U.L, E.CON.X.I.M.L, E.CON.X.I.C.L:
                  m ← c || d
      endcase
      case op of
            E.CON.X.I.U.B, E.CON.X.I.U.L:
                  as ← ms ← bs ← false
            E.CON.X.I.M.B, E.CON.X.I.M.L:
```

```
                ms ← false
                as ← bs ← true
            E.CON.X.I.B, E.CON.X.I.L, E.CON.X.I.C.B, E.CON.X.I.C.L:
                as ← ms ← bs ← true
        endcase
        h ← (2*gsize) + 7 - lgsize
        r ← h - size - sh
        for i ← 0 to wsize-gsize by gsize
            q[0] ← 0^(2*gsize+7-lgsize)
            for j ← 0 to vsize-gsize by gsize
                case op of
                    E.CON.X.I.B, E.CON.X.I.L, E.CON.X.I.M.B, E.CON.X.I.M.L,
                    E.CON.X.I.U.B, E.CON.X.I.U.L:
                        q[j+gsize] ← q[j] + mul(gsize,h,ms,m,i+128-j,bs,b,j)
                    E.CON.X.I.C.B, E.CON.X.I.C.L:
                        if (~i) & j & gsize = 0 then
                            q[j+gsize] ← q[j] + mul(gsize,h,ms,m,i+128-j,bs,b,j)
                        else
                            q[j+gsize] ← q[j] - mul(gsize,h,ms,m,i+128-j+2*gsize,bs,b,j)
                        endif
                endcase
            endfor
            p ← q[vsize]
            case rnd of
                none, N:
                    s ← 0^(h-r) || ~p_r || p_r^(r-1)
                Z:
                    s ← 0^(h-r) || p_(h-1)^r
                F:
                    s ← 0^h
                C:
                    s ← 0^(h-r) || 1^r
            endcase
            v ← ((as & p_(h-1))||p) + (0||s)
            if (v_(h..r+gsize) = (as & v_(r+gsize-1))^(h+1-r-gsize) then
                a_(gsize-1+i..i) ← v_(gsize-1+r..r)
            else
                a_(gsize-1+i..i) ← as ? (v_h || ~v_h^(gsize-1)) : 1^gsize
            endif
        endfor
        a_(127..wsize) ← 0
        RegWrite(rd, 128, a)
enddef
```

Exceptions

none

Ensemble Convolve Floating-point

These instructions take an address from a general register to fetch a large operand from memory, a second operand from a general register, perform a group of operations on partitions of bits in the operands, and catenate the results together, placing the result in a general register.

Operation codes

E.CON.F.16.B	Ensemble convolve floating-point half big-endian
E.CON.F.16.L	Ensemble convolve floating-point half little-endian
E.CON.F.32.B	Ensemble convolve floating-point single big-endian
E.CON.F.32.L	Ensemble convolve floating-point single little-endian
E.CON.F.64.B	Ensemble convolve floating-point double big-endian
E.CON.F.64.L	Ensemble convolve floating-point double little-endian
E.CON.C.F.16.B	Ensemble convolve complex floating-point half big-endian
E.CON.C.F.16.L	Ensemble convolve complex floating-point half little-endian
E.CON.C.F.32.B	Ensemble convolve complex floating-point single big-endian
E.CON.C.F.32.L	Ensemble convolve complex floating-point single little-endian
E.CON.C.F.64.B	Ensemble convolve complex floating-point double big-endian
E.CON.C.F.64.L	Ensemble convolve complex floating-point double little-endian

Format

E.op.size.order rd=rc,rb

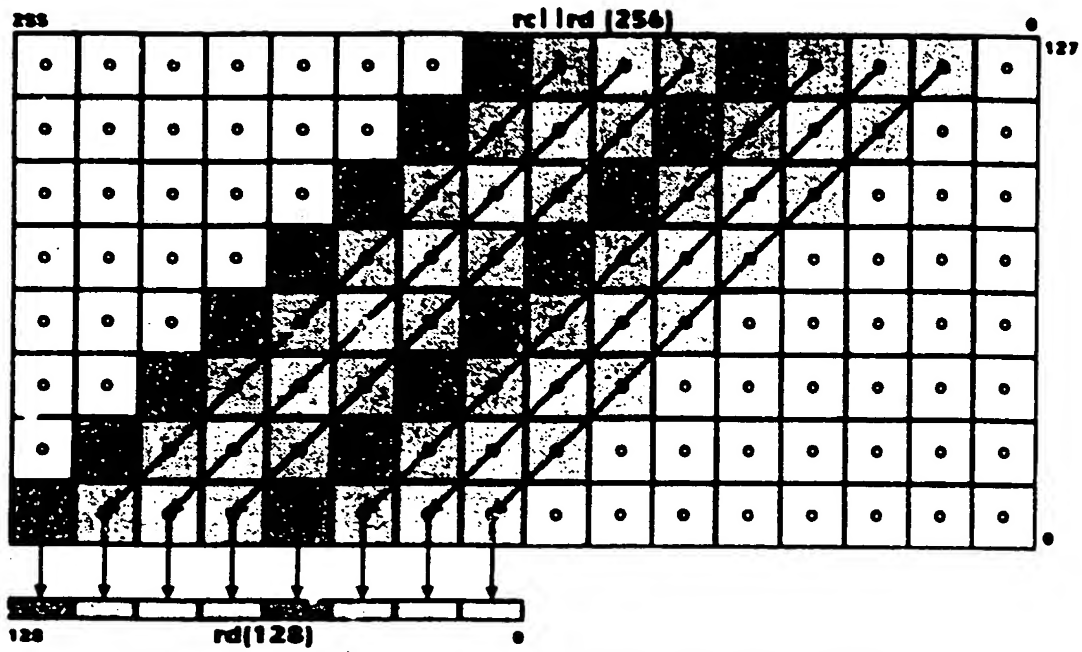
rd=eopsizeorder(rd,rc,rb)

31 24	23 18	17 12	11 6	5 0
E.size	rd	rc	rb	op.order
8	6	6	6	6

Description

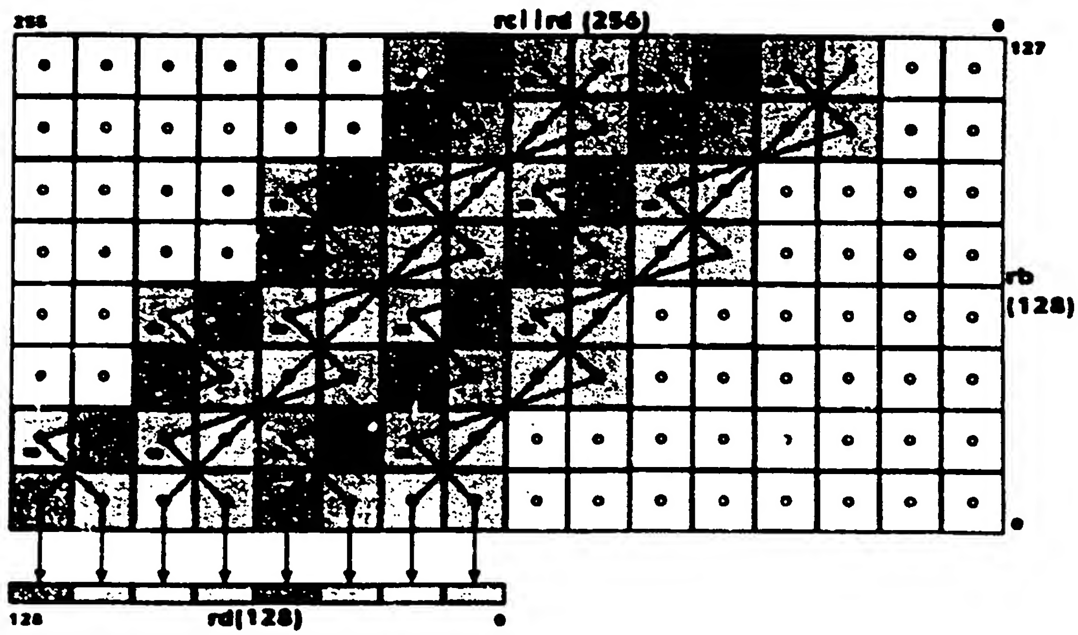
The first value is the catenation of the contents of register rd and rc, as specified by the order parameter. A second value is the contents of register rb. The values are partitioned into groups of operands of the size specified. The second values are multiplied with the first values, then summed, producing a group of result values. The group of result values is catenated and placed in register rd.

An ensemble-convolve-floating-point-half-little-endian instruction (E.CON.F.16.L) convolves vector [x w v u t s r q p o n m l k j i] with vector [h g f e d c b a], yielding the products [ax+bw+cv+du+et+fs+gr+hq ... as+br+cq+dp+eo+fn+gm+hl ar+bq+cp+do+en+fm+gl+hk aq+bp+co+dn+em+fl+gk+hj]:



Ensemble convolve floating-point half little-endian

A ensemble-convolve-complex-floating-point-half-little-endian instruction (E.CON.C.F.16.L) convolves vector [x w v u t s r q p o n m l k j i] with vector [h g f e d c b a], yielding the products [ax+bw+cv+du+et+fs+gr+hq ... as-bt+cq-dr+eo-fp+gm-hn ar+bq+cp+do+en+fm+gl+hk aq-br+co-dp+em-fn+gk+hl]:



Ensemble convolve complex floating-point half little-endian

Definition

```
def mul(size,v,i,w,j) as
    mul ← fmul(F(size,v_{size-1+i..i}),F(size,w_{size-1+j..j}))
enddef

def EnsembleConvolveFloatingPoint(op,gsize,rd,rc,rb)
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    lgsize ← log(gsize)
    wsize ← 128
    msize ← 256
    vsize ← 128
    case op of
        E.CON.F.B, E.CON.C.F.B:
            m ← d || c
        E.CON.F.L, E.CON.C.F.L :
            m ← c || d
    endcase
    for i ← 0 to wsize-gsize by gsize
        //NULL value doesn't combine with zero to alter sign bit
        q[0].t ← NULL
        for j ← 0 to vsize-gsize by gsize
            case op of
                E.CONF.L, E.CONF.B:
```

```
                    q[j+gsize] ← fadd(q[j], mul(gsize,m,i+128-j,b,j))
                E.CONCF.L, E.CONCF.B:
                    if (~i) & j & gsize = 0 then
                        q[j+gsize] ← fadd(q[j], mul(gsize,m,i+128-j,b,j))
                    else
                        q[j+gsize] ← fsub(q[j], mul(gsize,m,i+128-j+2*gsize,b,j))
                    endif
            endcase
        endfor
        a_{gsize-1+i..i} ← PackF(gsize,q[vsize],N)
    endfor
    a_{127..wsize} ← 0
    RegWrite(rd, 128, a)
enddef
```

Exceptions

none

Ensemble Extract

These operations take operands from three registers, perform operations on partitions of bits in the operands, and place the concatenated results in a fourth register.

Operation codes

E.MULX	Ensemble multiply extract
E.EXTRACT	Ensemble extract
E.SCALADD.X	Ensemble scale add extract

Format

E.opra=rd,rc,rb

ra=gop(rd,rc,rb)

31 … 24	23 … 18	17 … 12	11 … 6	5 … 0
op	rd	rc	rb	ra
8	6	6	6	6

Description

The contents of registers rd, rc, and rb are fetched. The specified operation is performed on these operands. The result is placed into register ra.

Bits 31..0 of the contents of register rb specifies several parameters which control the manner in which data is extracted, and for certain operations, the manner in which the operation is performed. The position of the control fields allows for the source position to be added to a fixed control value for dynamic computation, and allows for the lower 16 bits of the control field to be set for some of the simpler extract cases by a single GCOPYI.128 instruction. The control fields are further arranged so that if only the low order 8 bits are non-zero, a 128-bit extraction with truncation and no rounding is performed.

31 … 24	23 … 16	15	14	13	12	11	10 … 9	8 … 0
fsize	dpos	x	s	n	m	l	rnd	gssp
8	8	1	1	1	1	1	2	9

The table below describes the meaning of each label:

label	bits	meaning
fsize	8	field size
dpos	8	destination position
x	1	reserved
s	1	signed vs. unsigned
n	1	complex vs. real multiplication
m	1	merge vs. extract or mixed-sign vs. same-sign multiplication
l	1	limit: saturation vs. truncation
rnd	2	rounding
gssp	9	group size and source position

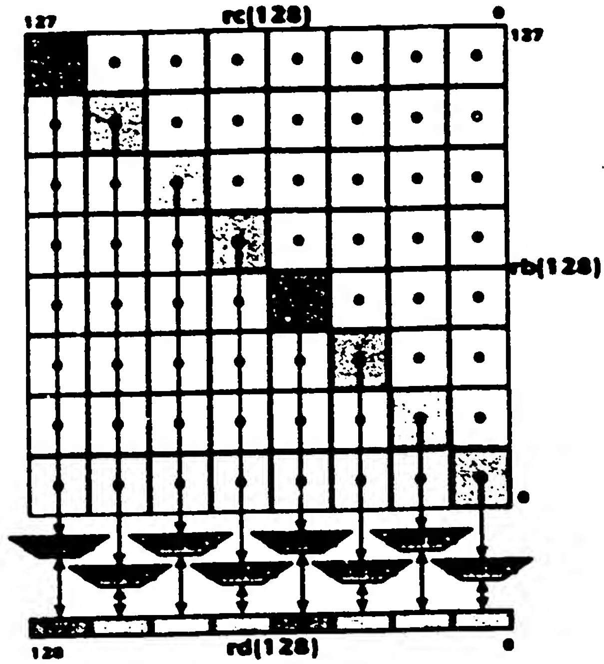
The 9-bit **gssp** field encodes both the group size, **gsize**, and source position, **spos**, according to the formula **gssp** = 512-4***gsize**+**spos**. The group size, **gsize**, is a power of two in the range 1..128. The source position, **spos**, is in the range 0..(2***gsize**)-1.

The values in the **s**, **n**, **m**, **l**, and **rnd** fields have the following meaning:

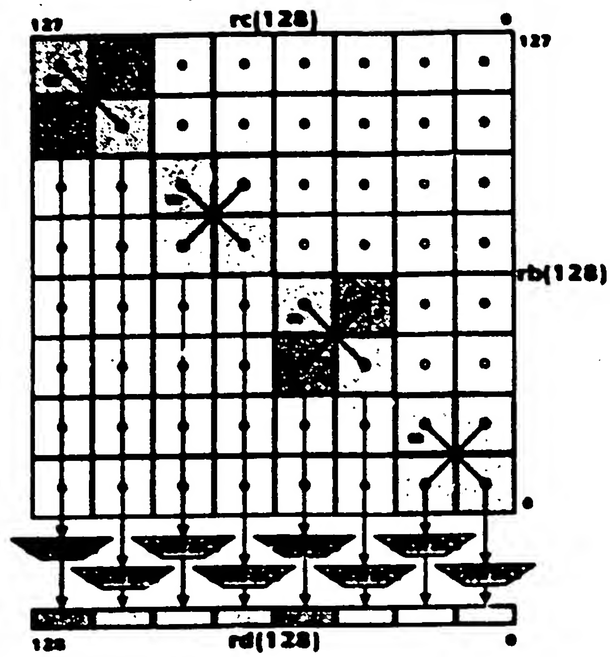
values	s	n	m	l	rnd
0	unsigned	real	extract/same-sign	truncate	F
1	signed	complex	merge/mixed-sign	saturate	Z
2					N
3					C

For the E.SCAL.ADD.X instruction, bits 127..64 of the contents of register rc specifies the multipliers for the multiplicands in registers ra and rb. Specifically, bits 64+2*gsize-1..64+gsize is the multiplier for the contents of register ra, and bits 64+gsize-1..64 is the multiplier for the contents of register rb.

An ensemble-multiply-extract-doublets instruction (E.MULX) multiplies vector ra [h g f e d c b a] with vector rb [p o n m l k j i], yielding the result [hp go fn em dl ck bj ai], rounded and limited as specified by $rc_{31..0}$.

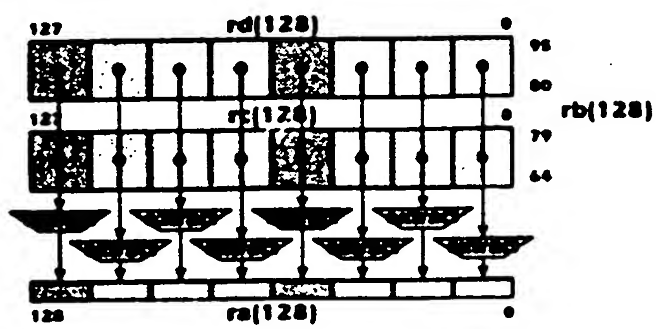


An ensemble-multiply-extract-doublets-complex instruction (E.MUL.X with n set) multiplies operand [h g f e d c b a] by operand [p o n m l k j i], yielding the result [gp+ho go-hp en+fm em-fn cl+dk ck-dl aj+bi ai-bj], rounded and limited as specified. Note that this instruction prefers an organization of complex numbers in which the real part is located to the right (lower precision) of the imaginary part.:



Ensemble complex multiply extract doublets

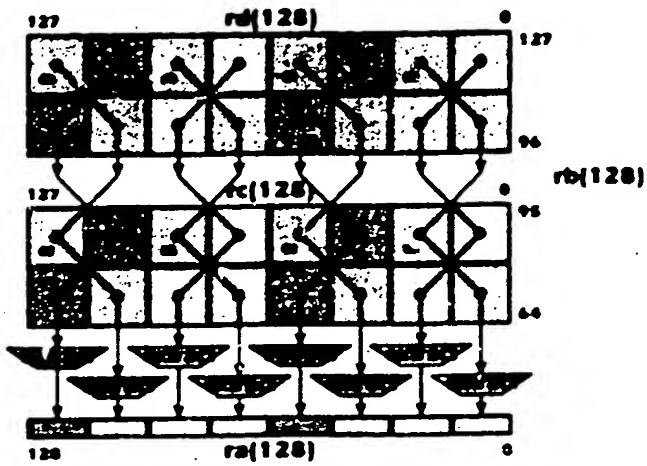
An ensemble-scale-add-extract-doublets instruction (E.SCAL.ADD.X) multiplies vector ra [h g f e d c b a] with rc95..80 [r] and adds the product to the product of vector rb [p o n m l k j i] with rc79..64 [q], yielding the result [hr+pq gr+oq fr+nq er+mq dr+lq cr+kq br+jq ar+iq], rounded and limited as specified by rc31..0.



Ensemble scale add extract doublets

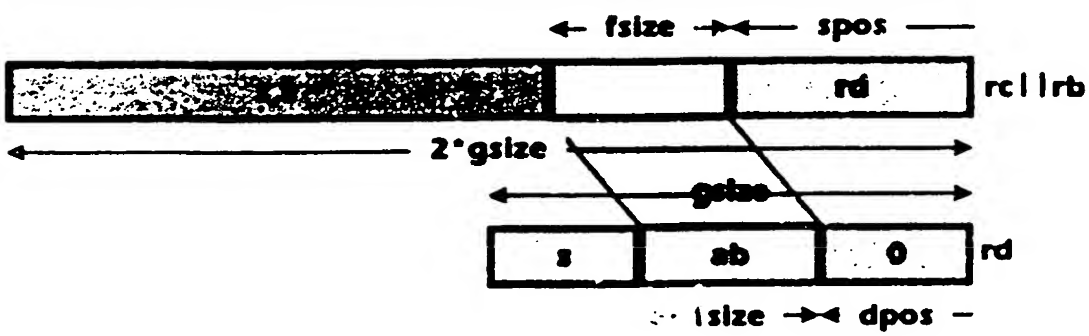
An ensemble-scale-add-extract-doublets-complex instruction (E.SCLADD.X with n set) multiplies vector ra [h g f e d c b a] with rc127..96 [t s] and adds the product to the product of vector rb [p o n m l k j i] with rc95..64 [r q], yielding the result [hs+gt+pq+or gs-ht+oq-pr

fs+et+nq+mr es-ft+mq-nr ds+ct+lq+kr cs-dt+kq-lr bs+at+jq+ir as-bt+iq-jr], rounded and limited as specified by $rc_{31..0}$.



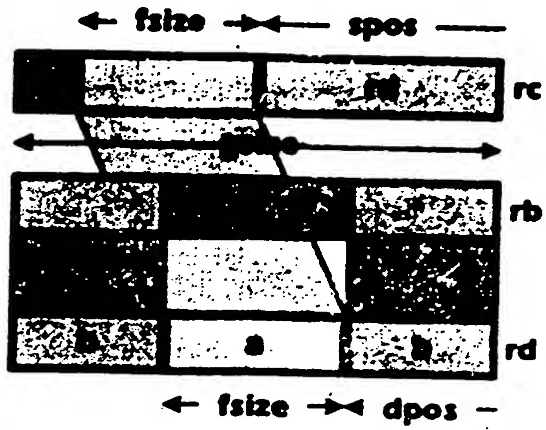
Ensemble complex scale add extract doublets

For the E.EXTRACT instruction, when m=0, the parameters are interpreted to select a fields from the catenated contents of registers rd and rc, extracting values which are catenated and placed in register ra.:



Ensemble extract

For an ensemble-merge-extract (G.X when m=1), the parameters are interpreted to merge a fields from the contents of register rd with the contents of register rc. The results are catenated and placed in register ra.



Ensemble merge extract

Definition

```
def mul(size,h,vs,v,i,ws,w,j) as
      mul ← ((vs&v_size-1+i)^(h-size) || v_size-1+i..i) * ((ws&w_size-1+j)^(h-size) || w_size-1+j..j)
enddef

def EnsembleExtract(op,ra,rb,rc,rd) as
      d ← RegRead(rd, 128)
      c ← RegRead(rc, 128)
      b ← RegRead(rb, 128)
      case b8..0 of
            0..255:
                  sgsize ← 128
            256..383:
                  sgsize ← 64
            384..447:
                  sgsize ← 32
            448..479:
                  sgsize ← 16
            480..495:
                  sgsize ← 8
            496..503:
                  sgsize ← 4
            504..507:
                  sgsize ← 2
            508..511:
                  sgsize ← 1
      endcase
      l ← b11
      m ← b12
      n ← b13
      signed ← b14
```

```
case op of
    E.EXTRACT:
        gsize ← sgsize
        h ← (2-m)*gsize
        as ← signed
        spos ← (b8..0) and ((2-m)*gsize-1)
    E.SCALADD.X:
        if (sgsize < 8) then
            gsize ← 8
        elseif (sgsize*(n+1) > 32) then
            gsize ← 32/(n+1)
        else
            gsize ← sgsize
        endif
        ds ← cs ← signed
        bs ← signed ^ m
        as ← signed or m or n
        h ← (2*gsize) + 1 + n
        spos ← (b8..0) and (2*gsize-1)
    E.MUL.X:
        if (sgsize < 8) then
            gsize ← 8
        elseif (sgsize*(n+1) > 128) then
            gsize ← 128/(n+1)
        else
            gsize ← sgsize
        endif
        ds ← signed
        cs ← signed ^ m
        as ← signed or m or n
        h ← (2*gsize) + n
        spos ← (b8..0) and (2*gsize-1)
endcase
dpos ← (0 || b23..16) and (gsize-1)
r ← spos
sfsize ← (0 || b31..24) and (gsize-1)
tfsize ← (sfsize = 0) or ((sfsize+dpos) > gsize) ? gsize-dpos : sfsize
fsize ← (tfsize + spos > h) ? h - spos : tfsize
if (b10..9 = Z) and not as then
    rnd ← F
else
    rnd ← b10..9
endif
for i ← 0 to 128-gsize by gsize
    case op of
        E.EXTRACT:
            if m then
                p ← dgsize+i-1..i
            else
                p ← (d || c)2*(gsize+i)-1..2*i
            endif
        E.MULX:
            if n then
```

```
                    if (i and gsize) = 0 then
                        p ← mul(gsize,h,ds,d,i,cs,c,i) - mul(gsize,h,ds,d,i+size,cs,c,i+size)
                    else
                        p ← mul(gsize,h,ds,d,i,cs,c,i+size) + mul(gsize,h,ds,d,i,cs,c,i+size)
                    endif
                else
                    p ← mul(gsize,h,ds,d,i,cs,c,i)
                endif
            E.SCALADD.X:
                if n then
                    if (i and gsize) = 0 then
                        p ← mul(gsize,h,ds,d,i,bs,b,64+2*gsize)
                            + mul(gsize,h,cs,c,i,bs,b,64)
                            - mul(gsize,h,ds,d,i+gsize,bs,b,64+3*gsize)
                            - mul(gsize,h,cs,c,i+gsize,bs,b,64+gsize)
                    else
                        p ← mul(gsize,h,ds,d,i,bs,b,64+3*gsize)
                            + mul(gsize,h,cs,c,i,bs,b,64+gsize)
                            + mul(gsize,h,ds,d,i+gsize,bs,b,64+2*gsize)
                            + mul(gsize,h,cs,c,i+gsize,bs,b,64)
                    endif
                else
                    p ← mul(gsize,h,ds,d,i,bs,b,64+gsize) + mul(gsize,h,cs,c,i,bs,b,64)
                endif
        endcase
        case rnd of
            N:
                s ← 0^(h-r) || ~p_r || p_r^(r-1)
            Z:
                s ← 0^(h-r) || p^r_(h-1)
            F:
                s ← 0^h
            C:
                s ← 0^(h-r) || 1^r
        endcase
        v ← ((as & p_(h-1))||p) + (0||s)
        if (v_(h..r+fsize) = (as & v_(r+fsize-1))^(h+1-r-fsize)) or not (l and (op = E.EXTRACT)) then
            w ← (as & v_(r+fsize-1))^(gsize-fsize-dpos) || v_(fsize-1+r..r) || 0^dpos
        else
            w ← (s ? (v_h || ~v_h^(gsize-dpos-1)) : 1^(gsize-dpos)) || 0^dpos
        endif
        if m and (op = E.EXTRACT) then
            a_(size-1+i..i) ← c_(gsize-1+i..dpos+fsize+i) || w_(dpos+fsize-1..dpos) || c_(dpos-1+i..i)
        else
            a_(size-1+i..i) ← w
        endif
    endfor
    RegWrite(ra, 128, a)
enddef
```

Exceptions

none

Ensemble Extract Immediate

These operations take operands from two registers and a short immediate value, perform operations on partitions of bits in the operands, and place the concatenated results in a third register.

Operation codes

E.EXTRACT.I.8.C	Ensemble extract immediate signed bytes ceiling
E.EXTRACT.I.8.F	Ensemble extract immediate signed bytes floor
E.EXTRACT.I.8.N	Ensemble extract immediate signed bytes nearest
E.EXTRACT.I.8.Z	Ensemble extract immediate signed bytes zero
E.EXTRACT.I.16.C	Ensemble extract immediate signed doublets ceiling
E.EXTRACT.I.16.F	Ensemble extract immediate signed doublets floor
E.EXTRACT.I.16.N	Ensemble extract immediate signed doublets nearest
E.EXTRACT.I.16.Z	Ensemble extract immediate signed doublets zero
E.EXTRACT.I.32.C	Ensemble extract immediate signed quadlets ceiling
E.EXTRACT.I.32.F	Ensemble extract immediate signed quadlets floor
E.EXTRACT.I.32.N	Ensemble extract immediate signed quadlets nearest
E.EXTRACT.I.32.Z	Ensemble extract immediate signed quadlets zero
E.EXTRACT.I.64.C	Ensemble extract immediate signed octlets ceiling
E.EXTRACT.I.64.F	Ensemble extract immediate signed octlets floor
E.EXTRACT.I.64.N	Ensemble extract immediate signed octlets nearest
E.EXTRACT.I.64.Z	Ensemble extract immediate signed octlets zero
E.EXTRACT.I.U.8.C	Ensemble extract immediate unsigned bytes ceiling
E.EXTRACT.I.U.8.F	Ensemble extract immediate unsigned bytes floor
E.EXTRACT.I.U.8.N	Ensemble extract immediate unsigned bytes nearest
E.EXTRACT.I.U.16.C	Ensemble extract immediate unsigned doublets ceiling
E.EXTRACT.I.U.16.F	Ensemble extract immediate unsigned doublets floor
E.EXTRACT.I.U.16.N	Ensemble extract immediate unsigned doublets nearest
E.EXTRACT.I.U.32.C	Ensemble extract immediate unsigned quadlets ceiling
E.EXTRACT.I.U.32.F	Ensemble extract immediate unsigned quadlets floor
E.EXTRACT.I.U.32.N	Ensemble extract immediate unsigned quadlets nearest
E.EXTRACT.I.U.64.C	Ensemble extract immediate unsigned octlets ceiling
E.EXTRACT.I.U.64.F	Ensemble extract immediate unsigned octlets floor
E.EXTRACT.I.U.64.N	Ensemble extract immediate unsigned octlets nearest
E.MULX.I.8.C	Ensemble multiply extract immediate signed bytes ceiling
E.MULX.I.8.F	Ensemble multiply extract immediate signed bytes floor
E.MULX.I.8.N	Ensemble multiply extract immediate signed bytes nearest
E.MULX.I.8.Z	Ensemble multiply extract immediate signed bytes zero
E.MULX.I.16.C	Ensemble multiply extract immediate signed doublets ceiling
E.MULX.I.16.F	Ensemble multiply extract immediate signed doublets floor
E.MULX.I.16.N	Ensemble multiply extract immediate signed doublets nearest
E.MULX.I.16.Z	Ensemble multiply extract immediate signed doublets zero
E.MULX.I.32.C	Ensemble multiply extract immediate signed quadlets ceiling
E.MULX.I.32.F	Ensemble multiply extract immediate signed quadlets floor